ENABLING SCALABLE SELF-MANAGEMENT FOR ENTERPRISE-SCALE SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Vibhore Kumar

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2008

ENABLING SCALABLE SELF-MANAGEMENT FOR ENTERPRISE-SCALE SYSTEMS

Approved by:

Prof. Karsten Schwan,
Committee Chair
College of Computing
Georgia Institute of Technology

Prof. Nick Feamster
College of Computing
Georgia Institute of Technology

Prof. Karsten Schwan, Advisor
College of Computing
Georgia Institute of Technology

Prof. Ling Liu
College of Computing
Georgia Institute of Technology

Dr. Brian F. Cooper
Yahoo! Research

Dr. Akhil Sahai
VMware Inc.

Date Approved: 27 March 2008

To my mother, Swatantra Srivastava.

ACKNOWLEDGEMENTS

I may have written this dissertation but there is much more to it than just the writing, there are people, without whose help this dissertation would not have been possible. There are people who, all the way to this dissertation, have believed in me, inspired me, supported me and questioned me, and I would like to make the most of this opportunity to thank them all.

First of all, I would like to thank my advisor Prof. Karsten Schwan. Prof. Schwan not only listened patiently to my often 'over-ambitious' research plans but also got involved in them, added reason and gave me the freedom to pursue the plans. Working under his guidance has been a pleasure and has been everything I thought Ph.D. research would be. I would also like to thank Dr. Brian F. Cooper whose contribution to this dissertation has been nothing less than that of an advisor. Dr. Cooper taught me the discipline of taking small steps, addressing one problem at a time. He would not solve the problems for me but teach me ways to solve them and I am really grateful to him for the investment he made in me.

I would also like to thank Prof. Ling Liu and Prof. Nick Feamster for their insightful comments during the various stages of my research that have helped shape the course of this dissertation. I owe a special thanks to Dr. Greg Eisenhauer, Dr. Matthew Wolf, Dr. Ada Gavrilovska, Dr. David M. Chess, Dr. Rob Strom, Dr. Akhil Sahai, Dr. Subu Iyer and Dr. Yuan Chen for sharing their experience, thoughts and comments about my research. A good bunch of friends has helped me endure and enjoy the ups and downs of Ph.D. life and Ankur, Nandita, Sandip, Sangeetha, Sanjay, Srihari, Sunitha and Vikas deserve a special mention.

Finally, I would like to thank my parents R. K. Srivastava and Swatantra Srivastava and brothers Vaibhav and Vipul for the encouragement and love that has always left me inspired. Most importantly, my wife, Smita deserves a special mention and thanks for tolerating my absent-mindedness and blank looks when I was happily treading the course to this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Today's Enterprise IT infrastructures, systems and applications are more distributed, more dynamic, more inter-related, and more complex than ever. Certainly, often, this has accelerated the pace at which the business is conducted by such enterprises but it has also introduced substantial complexity into enterprise IT fabrics. The fallout from this growing complexity is the increasing inability of human administrators to manage systems and keep them operational, which can translate to additional downtime or unexpected outages, violation of Service-Level Agreements (SLAs), and failure to comply with regulations. One outcome has been new industry initiatives to increase the levels of automation and improve system manageability, termed 'Autonomic Computing', 'Adaptive Enterprise', 'Dynamic Systems Initiative', and similar. Their common goal is to build systems that (1) provide primitives that facilitate the implementation of self-management capabilities, and (2) use automated methods to manage themselves based on higher level goals or directives.

Implementing self-management for enterprise systems is difficult. First, the scale and complexity of such systems makes it hard to understand and interpret system behavior or worse, the root causes of certain behaviors. Second, it is not clear how the goals specified at a system-level translate to component-level actions that drive the system. Third, the dynamic environments in which such systems operate requires self-management techniques that not only adapt the system but also adapt their own decision making processes. Finally, to build a self-management solution that is acceptable to administrators, it should have the properties of tractability and trust, which allow an administrator to both understand and fine-tune self-management actions.

This dissertation work introduces, implements, and evaluates *i*Manage, a novel

system state-space based framework for enabling self-management of enterprise-scale systems. The system state-space, in *i*Manage, is defined to be a collection of monitored system parameters and metrics (termed system variables). In addition, from amongst the system variables, it identifies the *variables of interest*, which determine the operational status of a system, and the *controllable variables*, which are the ones that can be deterministically modified to affect the operational status of a system. Using this formal representation, we have developed and integrated into *i*Manage techniques that establish a model relating the variables of interest and the controllable variables under the prevailing operational conditions. Such models are then used by *i*Manage in two distinct ways to facilitate self-management:

1. in the event of SLA violation, suggest an assignment of values to controllable variables (termed *action*) that ensures SLA compliance; and

2. determine per-component ranges for controllable variables, which if independently adhered to by each component, lead to SLA compliance.

To address the issue of scale in determining system models, *i*Manage makes use of a novel *state-space partitioning* scheme that partitions the state-space into smaller sub-spaces thereby allowing us to more precisely model the critical system aspects. *i*Manage uses probabilistic techniques to create *micro-models* for each partitioned sub-space, where such micro-models can evolve with time and encode the relationship between the variables of interest and the controllable variables. Our chosen modeling techniques are such that the generated models can be easily understood, augmented and modified by the administrator. Furthermore, *i*Manage associates each proposed self-management action with a *confidence-attribute* that determines whether the self-management action in question merits autonomic enforcement or not.

Beyond presenting the *i*Manage self-management framework, this dissertation also

describes our earlier work on enabling self-management for an important class of enterprise systems - the enterprise information systems used to drive the daily operations of large corporations. First, when using *Dynamic Overlays* to realize enterprise-scale data-flows, it is apparent that in critical enterprise settings like ours, the criteria that determine the utility of a certain flow or system configuration may not be attainable bandwidth or end-to-end delay, as assumed in both our earlier research and in most related work. Instead, it is net business-utility that is of primary concern to the enterprise. However, when using *Utility-driven Methods* for self-management, simple functional formulations of utility like those used in earlier work[8, 49] are difficult to obtain, as evident from both our past work in the real-time domain and from our interactions with industry collaborators at Delta Air Lines and Worldspan. The result is a need for new ways to align the governance of enterprise systems with higher-level business goals and directives. The *i*Manage framework does so by modeling the enterprise system and then using this model to determine from higher-level goals and directives the lower-level actions that can be used to achieve the goals, while keeping the administrator in the self-management loop.

The *i*Manage framework has been evaluated using both simulations and testbed experiments. Simulation experiments are carried out with a novel simulator capable of simulating distributed systems built with service oriented architecture (SOA). The simulator is equipped with functionality that allows it to create a distributed network of nodes and network-links, instantiate services on the nodes, and attach clients to the simulation that generate and consume events. The simulator can be run for a pre-specified amount of time, while recording the state-space variables at regular intervals. The controllable variables associated with the system being simulated can be changed at simulation run-time to enable self-management. Simulation experiments demonstrate that the techniques proposed as part of the *i*Manage framework can scale to systems with a large number of state-space variables and that the micro-models

constructed for sub-spaces are more accurate than monolithic models constructed for the entire state-space. Testbed experiments are conducted using the well-known application benchmark, RUBiS [60], with each component hosted on a different virtual machine. The experimental setup provides the capability to monitor the various state-space variables and the capability to modify the CPU and memory allocations of the VMs, in addition to some other application variables like load-balancing factor, number of threads, etc. Our experiments show that *i*Manage's techniques are not only able to determine the controllable variables to be modified in case of a SLA violation, but are also able to suggest, with some probabilistic guarantee, new values for such variables that will lead to SLA compliance.

CHAPTER I

INTRODUCTION

'If we don't get a handle on complexity, it will stop the expansion.'

- Paul Horn, Senior Vice President, IBM Research

It is not a revelation that computing systems are getting more and more complex, and if there exists something called the 'complexity wall', we are standing right beside it. The growing complexity of IT infrastructures has begun to hamper the growth of enterprises that have so far used them to expedite their growth. A recent survey by a leading technology and market research company found that enterprises now devote 80 percent of their overall IT spending to maintenance and ongoing operations. Innovation has taken a back seat and the enterprises are striving hard to keep the IT systems up and walking (running will be an exaggeration). It is becoming increasingly clear that administrators are having difficulties managing these systems, whose behavior can be best described as 'highly unpredictable'. To handle this complexity, solutions must be found that facilitate the automated management of such systems driven by high-level goals.

1.1 Enterprise Self-Management: The Issues

Implementing self-management for enterprise systems is difficult, and this research identifies four important problems that need to be addressed to enable scalable self-management for enterprise systems:

1. *The Problem of Scale* - the scale and complexity of enterprise systems makes it hard to understand and interpret system behavior or worse, the root causes of certain behaviors.

2. *The Problem of Complex System Modeling* - it is not clear how the goals specified at a higher level translate to lower-level actions that drive the system.

3. *The Problem of Dynamism* - the dynamic environments in which systems operate requires self-management techniques that not only adapt the system but also adapt their own decision making processes.

4. *The Problem of Tractability & Trust* - to make a self-management solution that is acceptable to administrators, it should have the properties of tractability and trust, which allow an administrator to both understand and fine-tune the self-management actions.

1.2 Background

The problem of self-management is of interest both to industry and academia. Industry efforts, which include the 'Autonomic Computing' initiative by IBM, the 'Adaptive Enterprise' program by HP, and the 'Dynamic Systems Initiative' led by Microsoft, are all targeted towards the ultimate goal of self-management. The increased activity by these industry majors has made some significant contributions to the self-management domain, which range from standardization of several interaction interfaces like error logs, web-services, etc., to the enhancement of management suites like IBM's Tivoli and HP's OpenView (now merged with Mercury)[1] with capabilities that facilitate easier problem diagnosis, to the augmentation of existing enterprise software components like databases and web-servers with certain self-configuration and self-optimization capabilities. There has also been a substantial amount of activity in academia to respond to growing system complexity by enabling technologies that assist in system self-management. Researchers have successfully embedded limited self-managing

[1]IBM, HP, Microsoft, Tivoli, OpenView and Mercury are registered trademarks of their respective owners.

capabilities into subsystems like database backends [72], request schedulers for multi-tier web services [15], and others. To complement such self-managing subsystems, researchers have devised policy-specification language [21], developed novel model building techniques [11] and created efficient monitoring schemes [4]. Similarly, there has been substantial progress in automating specific tasks that are required for enabling self-management. A particular effort of note is the work on automated problem diagnosis, presented in [19, 78]. The work focuses on using the monitoring data gathered from a system to detect service level objective violations and correlating the violation to earlier violations for gaining useful insights.

This dissertation contributes to the general domain of system management by combining specific contributions like those listed above into a comprehensive framework for managing complex systems. The remainder of this introduction describes in more detail its purpose, goals, approaches, and outcomes.

1.3 Thesis Statement

Self-Management can Scale to Large Distributed Enterprise Systems with Complex Goals & Constraints.

*1.4 Solution Approach: i*Manage

This dissertation proposes '*i*Manage' [48], a framework for enabling scalable self-management of enterprise systems. The goal of our research is to develop abstractions and methods that help bridge the gap between (i) the specific progress made in the general domain of self-management, like automation of well-defined subsystems or specialized techniques for certain self-management tasks vs. (ii) the more general challenges posed by managing more complex and/or larger IT infrastructures and applications. Toward this end, *i*Manage builds on such prior work for online system management, adopts the use of online monitoring and behavior detection tools and techniques [20], and endorses the use of Service Level Agreements (SLAs) for

specifying the higher level goals. However, to also address the broader management challenges posed by complex and dynamic IT applications and infrastructures, we propose a novel representation of the system state-space, and we develop new techniques for dealing with the problems of *scale*, *complex system modeling*, *dynamism*, *tractability* and *trust*. Here, tractability refers to an administrator's ability to understand current management actions undertaken by the system and to the system's ability to expose its reasoning for those actions.

1.5 Important Technical Contributions

To achieve the goal of system manageability the *i*Manage framework makes the following contributions -

- *A system modeling framework* - *i*Manage makes use of existing tools and techniques to collect system parameters and metrics (collectively called *system variables*), but then organizes them into a single representation of the system state-space and identifies which actions are available to change the system state.

- *A scheme for reducing the complexity of the system model* - since a typical system model is too complex to be used or even properly constructed, our tools provide mechanisms to partition the state-space into smaller units. These *micro-models* allow us to more precisely model critical aspects of the system, and to more effectively ensure conformance to service level agreements.

- *Ability to handle dynamic systems* - system models that are appropriate under one set of conditions may become invalid as operating conditions and/or the environment change. The multiple micro-models that constitute the system model in *i*Manage allow us to handle the heterogeneity in system behavior caused by varying operating conditions. Additionally, the micro-models can be more easily updated as compared to single monolithic models to handle

dynamism.

- *Techniques for quantifying our confidence in a system model* - in order for our system models to be useful, system administrator must be able to trust them. *i*Manage associates a confidence value with each proposed self-management action, and allows the administrator to both understand and use this confidence value when deciding whether to permit the system manage itself.

- *Techniques for determining component-level SLAs* - in general, managing a single component is easier than managing a system composed of several components. *i*Manage provides techniques that can be used to decompose a given system-level SLA to component-level SLAs [50], which if independently adhered to, leads to SLA compliance.

Several of the ideas that form a part of this dissertation have evolved from our earlier work on enabling self-management for enterprise information systems. Expressed as a natural sequence of events - the work started with the aim of enabling self-management for enterprise information systems, which resulted in our research on *dynamic data overlays* [46]. However, our further interaction with industry collaborators Delta Air Lines and Worldspan, which had motivated much of that work, made us realize that the enterprises are more interested in optimizing net-utility rather than traditional metrics like minimizing bandwidth utilization or reducing end-to-end delay, as assumed both in our earlier research and in much of the related work. To address the issue, we designed and implemented, *i*FLOW, a new *utility-driven middleware* [49, 47, 45] with well-defined abstractions for describing information-flows. The middleware also implemented the autonomic capabilities of self-configuration, self-optimization and self-healing [14], which were totally contained 'behind' the abstraction. The outcome was utility-driven middleware that relied on the assumption that one can express such a formulation for the system in question.

The *i*Manage framework substantially broadens the scope of our earlier work on self-management in enterprise systems, in part by removing the need for the precise formulation of system utility. This is important, because utility formulations are an encapsulation of the system model in mathematical terms, but unfortunately, it is not always possible to devise a useful and realistic formulation of utility for actual large-scale systems. This dissertation, therefore, also presents alternative methods for expressing high level objectives that can be decomposed into lower level ones upon which management methods can act.

1.6 Organization of This Dissertation

The remainder of this dissertation is organized as follows:

Chapter 2 describes our work on self-managing dynamic data overlays to instantiate and manage enterprise-wide information flows.

Chapter 3 presents the utility-driven approach to system self-management of enterprise information systems.

Chapter 4 describes the *i*Manage framework and its application to enabling SLA-based self-management.

Chapter 5 describes the use of *i*Manage framework to determine component-level SLAs from a given system-level SLA.

Chapter 6 concludes by providing details on the lessons that have been learnt during this research, and provides an outline for future work.

CHAPTER II

DYNAMIC DATA OVERLAYS

2.1 Introduction

Many emerging distributed applications must cope with a critical issue: how to efficiently aggregate, use, and make sense of the enormous amounts of data that is generated by these applications. Examples include sensor systems [54], distributed scientific processes like SkyServer [66], operational information systems used by large corporations [30], and others. Middleware initiatives for such applications include publish/subscribe systems like IBM's Gryphon [65] project or related academic endeavors [25], or commercial infrastructures based on web services, based on technologies like TPF, or using in-house solutions. However, middleware that relies on centralized approaches to data aggregation suffers from high communication overheads, lack of scalability, and unpredictably high processing workloads at central servers.

Our solution is to use in-network aggregation to reduce the load problems encountered in centralized approaches. This approach exploits the fact that data in these applications is usually routed using overlay networks, such that updates from distributed data sources arrive at their destination after traversing a number of intermediate overlay nodes. Each intermediate node can contribute some of its cycles towards processing the updates it is forwarding, the resulting advantage being the distribution of processing workload and a possible reduction in communication overhead involved in transmitting data updates.

In this chapter, we examine how to construct a distributed system for processing and aggregating streams of data. However, in order to set up such a system with nodes ready to contribute their resources for data processing, we must address several

challenges, including:

- *Ease of Deployment* - provide a simple interface for composing new streams from existing streaming data and also support run-time modifications to stream composition conditions.

- *Scalability* - there may be hundreds of streaming data sources, and the system should be incrementally scalable without significant overhead or effort.

- *Heterogeneity* - streaming data arrives at the sink after traversing a possibly heterogeneous set of nodes, which means that the system should support in-network processing of the streams at any node despite varying resource capabilities and operating environments.

- *Dynamism* - should automatically reconfigure to deal with changes in network conditions, node overloads and changes in data stream rates.

- *Performance* - since updates arrive at a very high rate, the infrastructure should not impose large overheads when aggregating and processing the updates.

We have implemented a Dynamic Data Overlay Infrastructure that makes it possible to compose new data streams by aggregating and processing existing streaming data originating at distributed locations. Our approach supports a SQL-like language to describe the data-flow graph for producing the new, transformed stream from existing data streams. The language allows users to refer to any stream originating in the system and supports attribute selection and join operations as in traditional databases. A resource-aware network-partitioning algorithm, described later, is used to assign operators from the flow graph to the underlying network nodes. The infrastructure relies on ECho [25], a pub-sub middleware developed at Georgia Tech, to deploy the data-operators for processing and forwarding the streaming updates in a heterogeneous environment. Automatic reconfiguration of stream overlays is

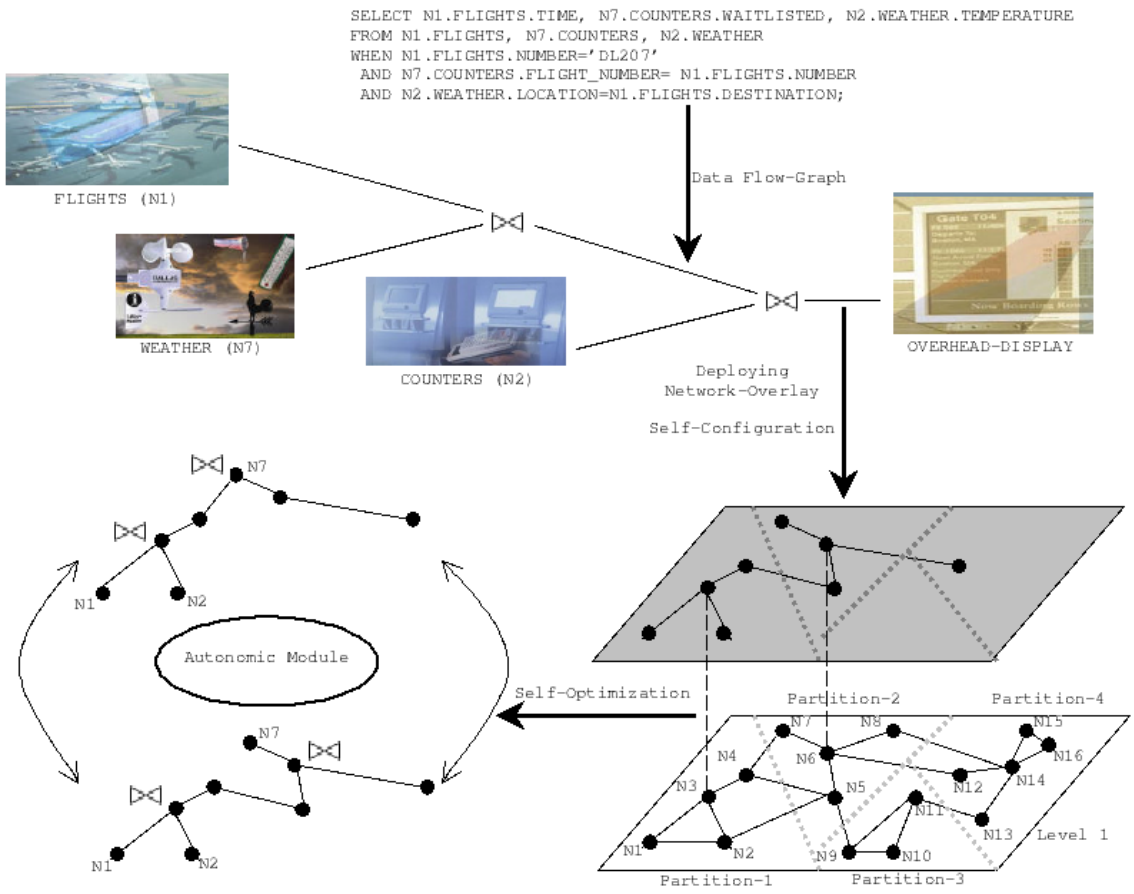

Figure 1: Overview of the Dynamic Data Overlay Approach

achieved by coupling the resource information collected from participating hosts with the Proactive Directory Service (PDS) [13], which is a subscription-based monitoring tool also developed by our group.

2.1.1 Delta's Operational Information System

An operational information system (OIS) [30] is a large-scale, distributed system that provides continuous support for a company's or organization's daily operations. One example of such a system we have been studying is the OIS run by Delta Air Lines, which provides the company with up-to-date information about all of its flight operations, including crews, passengers and baggage. Delta's OIS combines three different sets of functionality:

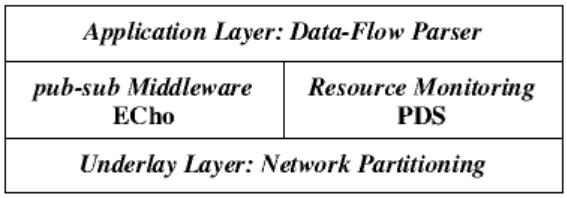

Figure 2: Three layered architecture of Dynamic Data Overlays

- *Continuous data capture* - for information like crew dispositions, passengers, airplanes and their current locations determined from FAA radar data.

- *Continuous status updates* - for low-end devices like airport flight displays, for the PCs used by gate agents, and even for large databases in which operational state changes are recorded for logging purposes.

- *Responses to client requests* - an OIS not only captures data and updates/distributes operational state, but it must also respond to explicit client requests such as pulling up information regarding bookings of a particular passenger. Certain clients may also generate additional state updates, such as changes in flights, crews or passengers.

2.2 Software Architecture

Our dynamic data overlay infrastructure is broadly composed of three layers as shown in Figure 4: (1) the Application Layer is responsible for accepting and parsing the data composition requests and constructing the data-flow graph, (2) the Middleware Layer consists of the ECho middleware and the PDS resource-monitoring infrastructure for deployment and maintenance of the stream overlay, and (3) the Underlay Layer organizes the nodes into hierarchical partitions that are used by the deployment infrastructure. The following subsections briefly describe these three layers.

2.2.1 Application Layer: Data-Flow Graph

Data flows are specified with our data-flow specification language. It closely follows the semantics and syntax of the SQL database language. The general syntax of our language is specified as follows

```
STREAM <attribute1> [<,attribute2> [<,attribute3> ]]
FROM <stream1> [<,stream2> [<,stream3> ]]
[WHEN <condition1> [<conjunction> <condition2>[]]]
```

In the data-flow specification language, the attribute list mentioned after the STREAM clause describes which components of each update are to be selected, the stream list following the FROM clause identifies the data stream sources, and finally, predicates are specified using the WHEN clause. Each stream in the infrastructure is addressable using the syntax *source_name.stream_name*. Likewise, an attribute in the stream is addressable using *source_name.stream_name.attribute*. Our language supports in-line operations on the attributes that are specified as $operator(attribute\_list[, parameter_list])$, where examples of such operations include SUM, MAX, MIN, AVG, $PRECISION$, etc. The system also provides the facility to extend this feature by adding user-defined operators.

The data-flow description is compiled to produce a data-flow graph. This graph consists of a set of operators to perform data transformations, as well as edges representing data streaming between operators. The graph is deployed in the network by assigning operators to network nodes.

2.2.2 Middleware Layer: ECho and PDS

The Middleware Layer supports the deployment and reconfiguration of the data-flow overlay. This support is provided by two components: ECho and PDS.

The ECho framework is a publish/subscribe middleware system that uses channel-based subscription (similar to CORBA). ECho streams data over stream channels,

which implement the edges between operators in the data-flow graph. The stream channels in our framework are not centralized; instead, they are lightweight distributed virtual entities managing data transmitted by middleware components at stream sources and sinks. The traffic for individual channels is multiplexed over shared communication links by aggregating the traffic of multiple streams into a single stream linking the two communicating addresses.

We follow the semantics of a publish-subscribe system in order to ensure that multiple sinks can subscribe/unsubscribe to a stream channel depending on their requirements, and that the channels survive even when there are no subscribers (although in that case no actual data is streamed). The publish-subscribe approach also proves useful when many sinks have similar data filtering needs; in such a scenario, a single channel derived using a data transformation operator can fill the needs of all of the sinks.

The data-operator in our infrastructure is typically a snippet of code written in a portable subset of C called E-Code. This snippet is transported as a string to the node where it has to be deployed. At the target-node, the code snippet is parsed, and native code is generated. The implicit context in which the code is executed is a function declaration of the form:

int operator(<input type> in, <output type> out)

A return value of 1 causes the update to be submitted, while a return value of 0 causes the update to be discarded. The function body may also modify the input before copying it to the output. New flow-graphs may use the streams from existing operators, or they may cause operators to be created or updated to stream additional relevant data if necessary.

Network-wide resource availability information is managed by the Proactive Directory Service (PDS). This information allows us to dynamically reconfigure the

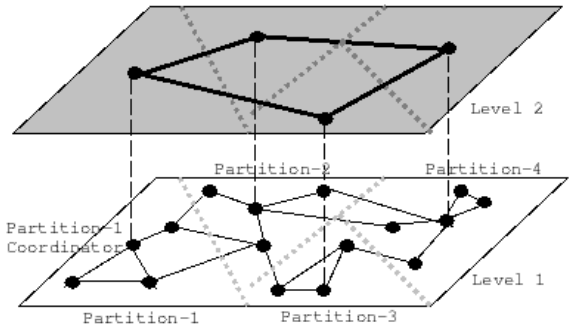

Figure 3: Hierarchical network partitioning

data-flow deployment in response to changing resource conditions. PDS is an efficient and scalable information repository with an interface that includes a proactive, push-based access mode. Through this interface, PDS clients can learn about objects (or types of objects) inserted in/removed from their environment and about changes to pre-existing objects. The infrastructure uses PDS objects to receive resource updates from the system when operating conditions change.

2.2.3 Underlay Layer: Network Partitioning

This layer is responsible for maintaining a hierarchy of physical nodes in order to cluster nodes that are close in the network sense, based on measures like end-to-end delay, bandwidth or inter-node traversal cost (a combination of bandwidth and delay). The hierarchy is used for network-aware deployment of the data-flow graph. Each node in a cluster knows about the costs of paths between each pair of nodes in the cluster. A node is chosen from each cluster to act as the coordinator for this cluster in the next level of the hierarchy. Like the physical nodes in the first level of hierarchy, the coordinator nodes can also be clustered to add another level in the hierarchy; similar to the initial level all the coordinators at a particular level know about average min cost path to the other coordinator nodes that fall in the same partition at that level. An example is shown in Figure 5.

The advantage of organizing nodes in a hierarchy is that it simplifies maintenance

of the clustering structure, and provides a simplified abstraction of the underlying network to the upper layers. Then, we can subdivide the data-flow graph to the individual clusters for further deployment. In order to scalably cluster nodes, we bound the amount of non-local information maintained by nodes by limiting the number of nodes that are allowed per cluster.

2.3 Deployment & Dynamic Reconfiguration

This section formally describes the data-flow graph deployment problem and it presents a highly scalable distributed algorithm that can be used to obtain an efficient solution to this problem. Then, we extend the deployment algorithm by incorporating resource-awareness.

2.3.1 Problem Statement

We consider the underlying network as a graph $N(V_n, E_n)$, where vertices V_n represent the actual physical nodes and the network connections between the nodes are represented by the edges E_n. We further associate each edge e_{ni} with a cost c_i that represents the application-oriented cost of traversing the corresponding network link. The data-flow graph derived from the SQL-like description is similarly represented as a graph $G(V_g, E_g)$ with each vertex in V_g representing a source-node, a sink-node or an operator, i.e.

$$V_g = V_{g-sources} \cup V_{g-sink} \cup V_{g-operators} \tag{1}$$

$V_{g-sources}$ is the set of stream sources for a particular data-flow graph and each source has a static association with a vertex in graph N. Source vertices have an associated update-rate. Vg-sink is the sink for the resulting stream of updates and it also has a static association with a vertex in graph N. $V_{g-operators}$ is the set of operators that can be dynamically associated with any vertex in graph N. Each operator vertex is characterized by a resolution factor, which represents the increase or decrease in the data flow caused by the operator. In general, join operators, which combine multiple

streams, increase the amount of data-flow; while select operations, which filter data from a stream, result in a corresponding decrease. The edges in the data-flow graph may span multiple intermediate edges and nodes in the underlying network graph.

We want to produce a mapping M, which assigns each $v_{gj} \in V_{g-operators}$ to a $v_{ni} \in V_n$. Thus, M implies a corresponding mapping of edges in G to edges in N, such that each edge e_{gj-k} between operators v_{gj} and v_{gk} is mapped to the network edges along the lowest cost path between the network nodes that v_{gj} and v_{gk} are assigned to. We define $cost(M)$ as the sum of the costs of the network edges mapped to the edges in the data flow graph.

For example, consider a cost function that measures the end-to-end delay. If e_{gk} is determined by vertices v_{gi} and v_{gj}, which in turn are assigned to vertices v_{ni} and v_{nj} of the network graph N, then the cost corresponding to edge e_{gk} is the delay along the shortest path between the vertices v_{ni} and v_{nj}.

The problem is to construct the lowest cost mapping M between the edges E_g in G to edges E_n in N.

2.3.2 Distributed Deployment Algorithm

Now, we present a distributed algorithm for deploying the data-flow graph in the network. In a trivial scenario we could have a central planner assign operators to network nodes, but this approach will obviously not scale for very large networks, and the planner can become a central point of failure. Our partitioning-based approach deploys the data-flow graph in a more decentralized way. In particular, nodes in the network self-organize into a network-aware set of clusters, such that nodes in the same cluster have low latency. Then, we can use this partitioned structure to deploy the data-flow graph in a network-aware way, without having full knowledge of the delay between all pairs of network nodes.

The result is that an efficient mapping M is constructed recursively, using the

hierarchical structure of the underlay-layer. This mapping may not be optimal, since our approach trades guaranteed optimality for scalable deployment. However, since the deployment is network-aware, the mapping should have low cost. Experiments presented later demonstrate that our algorithm produces efficient deployments.

We now formalize the partitioning scheme described in Section 2.2.3. Let

$$\begin{aligned} n^i_{total} &= \text{total nodes at level } i \text{ of the hierarchy} \\ n_{critical} &= \text{maximum number of nodes per partition} \\ v^i_{nj} &= \text{coordinator node for node } v_{nj} \text{ at level } i \end{aligned}$$

Note that $v^0_{nj} = v_{nj}$ and that all the participants of a partition know about minimum cost path to all other nodes in the same partition. We bound the amount of path information that each node has to keep by limiting the size of the cluster using $n_{critical}$. A certain level i in the hierarchy is partitioned when $n^i_{total} > n_{critical}$. We consider the physical nodes to be located at level 1 of the partition hierarchy and actual network values are used to partition nodes at this level. For any other level i in the hierarchy the average inter-partition cost (i.e. end-to-end delay, bandwidth, etc.) from level i-1 are used for partitioning the coordinator nodes from the level i-1. The approximate cost between any two vertices v_{nj} and v_{nk} at any level i in the hierarchy can be determined using the following equations:

$$cost^i(v_{nj}, v_{nk}) = \begin{cases} cost(v^{i-1}_{nj}, v^{i-1}_{nk}) & \text{for } v_{nj} \neq v_{nk} \\ 0 & \text{for } v_{nj} = v_{nk} \end{cases}$$

and

$$cost(v_{nj}, v_{nk}) = cost^l(v_{nj}, v_{nk}) | v^{l-1}_{nj} \neq v^{l-1}_{nk} \wedge v^l_{nj} = v^l_{nk}$$

In simple words, the cost at level i between any two vertices v_{nj} and v_{nk} of N is 0 if the vertices have the same coordinator at level i-1, otherwise it is equal to the cost at some level l where the vertices have the same coordinator and do not share the same coordinator at level l-1.

The distributed deployment algorithm works as follows, the given data-flow graph $G(V_g, E_g)$ is submitted as input to the top-level (say level t) coordinators. We construct a set of possible node assignments at level t by exhaustively mapping all the vertices $V_{g-operators} \in V_g$ to the nodes at this level. The cost for each assignment is calculated and the assignment with lowest cost is chosen. This partitions the graph G into a number of sub-graphs each allocated to a node at level t and therefore to a corresponding cluster at level t-1. The sub-graphs are then again deployed in a similar manner at level t-1. This process continues till we reach level 1, which is the level at which all the physical nodes reside.

2.3.3 Reconfiguration

The overlay reconfiguration process takes advantage of two important features of our infrastructure: (1) that the nodes reside in clusters and (2) that only intra-cluster minimum cost analysis is required. These features allow us to limit the reconfiguration to within the cluster boundaries, which in turn makes reconfiguration a low-overhead process. An overlay can be reconfigured in response to a variety of events, which are reported to the first-level cluster-coordinators by the PDS. These events include change in network delays, change in available bandwidth, change in data-operator behavior (we call this operator profiling), available processing capability, etc. Since it is impractical to respond to all such events reported by the PDS, we set thresholds that should be reached to trigger a reconfiguration. For example, a cluster-coordinator may recalculate the minimum cost paths and redeploy the assigned sub-graphs when more than half the links in the cluster have reported change in end-to-end delay. However, setting such thresholds depends on the application-level requirement for resource-awareness. In the work on *i*FLOW, described in Chapter 3, we make use of utility-functions to establish a closer integration between the application-level requirements and the reconfiguration framework.

Note that reconfigurations in the Dynamic Data Overlay system described above are not lossless in terms of updates, and some updates and state may be lost during the process. This is acceptable for most of the streaming applications, which are able to tolerate some level of approximation and loss. However, as part of the work on *i*FLOW we were able to provide capabilities for achieving lossless reconfigurations.

2.4 Experiments

We ran a set of experiments to evaluate the performance of Dynamic Data Overlays. First, we ran microbenchmarks to examine specific features of our system. Then, we created an end-to-end setup for an application case study using real data from Delta Airlines OIS. Our results show that our system is effective at deploying and reconfiguring data-flow graphs for distributed processing of streaming data.

2.4.1 Experimental setup

The GT-ITM internetwork topology generator [76] was used to generate a sample Internet topology for evaluating our deployment algorithm. This topology represents a distributed OIS scattered across several locations. Specifically, we use the transit-stub topology for the ns-2 simulation by including one transit domain that resembles the backbone Internet and four stub domains that connect to transit nodes via gateway nodes in the sub domains. Each stub domain has 32 nodes and the number of total transit nodes is 128. Links inside a stub domain are 100Mbps. Links connecting stub and transit domains, and links inside a transit domain are 622Mbps, resembling OC-12 lines. The traffic inside the topology was composed of 900 CBR connections between sub domain nodes generated by cmu-scen-gen [18]. The simulation was carried out for 1800 seconds and snapshots capturing end-to-end delay between directly connected nodes were taken every 5 seconds. These are then used as inputs for our distributed deployment algorithm.

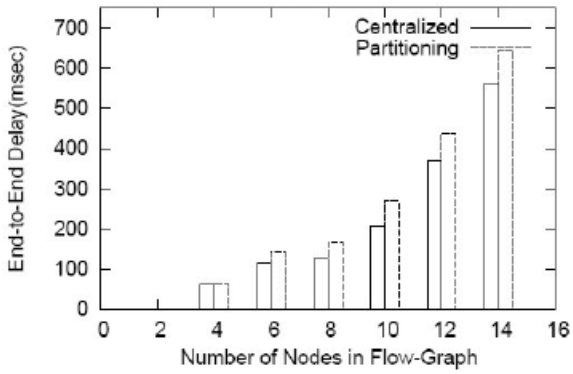

Figure 4: Comparison of end-to-end delay for centralized and partitioning approach

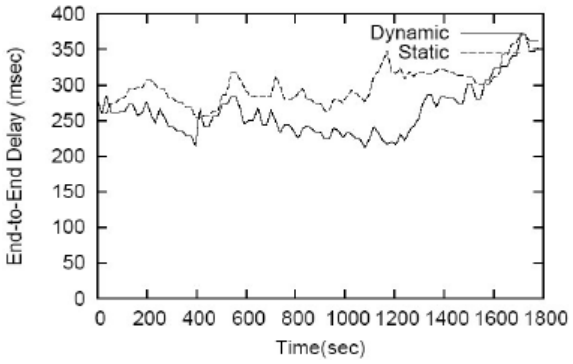

Figure 5: Variation of end-to-end delay with and without reconfiguration

2.4.2 Microbenchmarks

The first experiment focused on comparing the cost of a deployed data-flow graph using the centralized model as opposed to the partitioning based approach used in our infrastructure. Since in centralized approach we assume that a single node knows about minimum cost paths to all other nodes, the centralized approach gives the optimal deployment solution. However, the deployment time taken by centralized approach increases exponentially with the number of nodes in the network. Figure 4 shows that although the partitioned-based approach is not optimal, the cost of the deployed flow graph is not much worse than the deployment in the centralized approach, and is thus suitable for most scenarios.

The next experiment was conducted to examine the effectiveness of dynamic reconfiguration in providing an efficient deployment. Figure 5 shows the variation of end-to-end delay for a 10-node data-flow graph with changing network conditions, as simulated by introducing cross-traffic. The performance with dynamic reconfiguration is clearly better than with static deployment. It may be noted that at some points, cost of the dynamically reconfigured flow-graph becomes more than that of the static deployment. This happens because the cost calculation algorithm used in our approach calculates the graph cost that is an approximation of the actual deployment cost. In some cases the approximation is inaccurate, causing the reconfiguration to make a poor choice. However, these instances are rare, and when they do occur, the cost of the dynamic deployment is not much worse than the static deployment. Moreover, for most of the time dynamic reconfiguration produces a lower cost deployment.

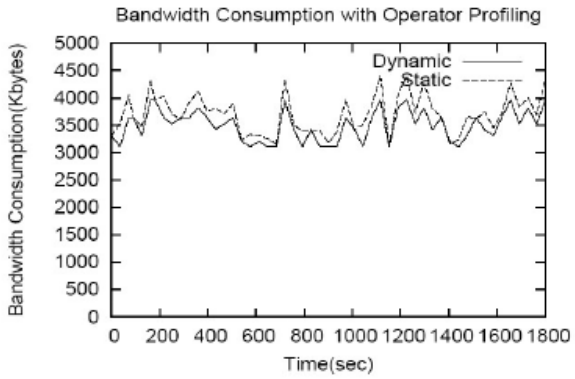

Figure 6: Variation in bandwidth consumption with dynamic reconfiguration and operator profiling

We also conducted experiments to compare the bandwidth consumption with and without dynamic reconfiguration. Each source was assumed to have a certain update rate of the form bytes/sec and each link was associated with a cost incurred per byte of data transferred using the link. Thus, at any point of time a deployed data graph has a cost, which is dependent on the links being used by the flow. We simulated a

change in resolution factor (the ratio of the amount of data flowing out versus flowing in) for each operator in the flow graph and measured the corresponding bandwidth utilization with dynamic reconfiguration and static deployment. Figure 6 shows that although dynamic reconfiguration helps in keeping the bandwidth consumption low, it does not offer very substantial gains; this is because when reconfiguration is driven by operator resolution it offers only a limited space for re-deployment.

2.4.3 Application case study

The next set of experiment was conducted on Emulab [26] with real data from the Delta OIS combined with simulated streams for Weather and News. The experiment was designed to emulate 4 different airport locations. The inter-location delays were set to 50ms while delays within an airport location were set to 2ms. The emulation was conducted with 13 nodes (Pentium-III, 850Mhz, 512MB RAM, RedHat Linux 7.1) and each location had only limited nodes connected to external locations. The experiment was motivated by the requirement to feed overhead displays at airports with up-to-date information. The overhead displays periodically update the weather and news at destination location and switch over to seating information for the aircraft at boarding gate. Other information displayed on such monitors includes names of wait-listed passengers, and current status of flight, etc. We deployed a flow graph with two operators, one for combining the weather and news information at destination and the other for selecting the appropriate flight data, which originates from a central location (Delta's TPF facility in this case).

The first experiment conducted on Emulab studied the behavior of system in case of network perturbation and then studied its response to processor overload. Once the data flow graph for providing an overhead display feed was deployed, we used iperf [38] to introduce traffic in some of the links used by the flow-graph. This is represented by the first spike in Figure 7. With dynamic reconfiguration the flow-graph responds

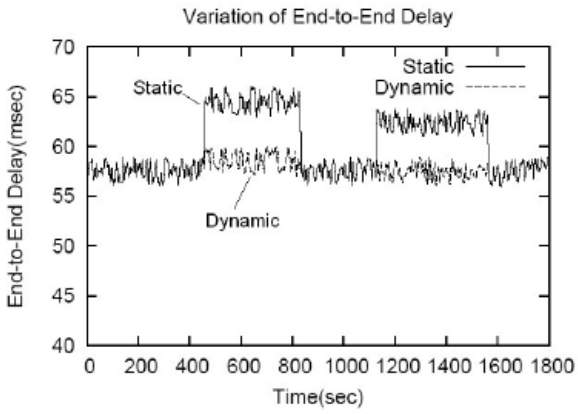

Figure 7: Variation of end-to-end delay for network perturbation and processor overload

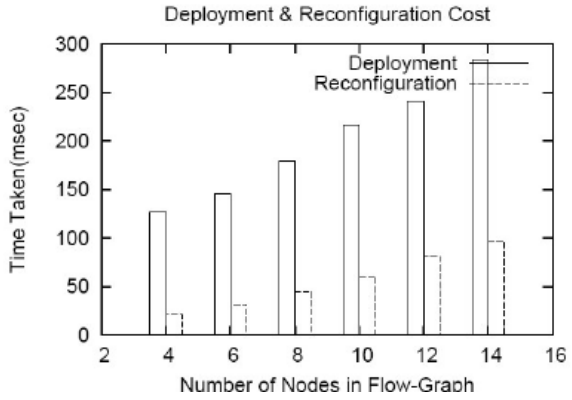

Figure 8: Comparison of deployment and reconfiguration cost on Emulab nodes

well to the spike in traffic; in contrast, the statically deployed graph experiences an increased delay. The next spike is a result of an increased processing load at both the operator nodes. Again with dynamic reconfiguration we end with a better delay than the static deployment. Even with dynamic reconfiguration the end-to-end delay spikes, but the time before the deployment adjusts is so short (milliseconds) that the spike is effectively unnoticeable.

The next experiment was conducted to compare the time for initial deployment and reconfiguration. Figure 8 shows that the times are quite small; only a few hundred milliseconds in the worst case. The figure illustrates the advantage of using a pub-sub

middleware for deploying the flow graph. The pub-sub channels have to be created only at the time of deployment; reconfiguration just involves a change in publisher and subscriber to this channel and is therefore even faster. It may also be noted that once the channels for the data-flow graph have been created, deployment is essentially a distributed process, which starts once the corresponding nodes receive the operator deployment messages. This makes deployment time to increase almost linearly with the number of nodes.

2.5 Related Work

The dynamic data overlays infrastructure is very closely associated with topics that are of interest to the middleware community, and to those interested in large-scale distributed data management.

2.5.1 Stream Processing & Distributed Databases

Data-stream processing has recently been an area of tremendous activity for database researchers; several groups such as STREAM [7] at Stanford, Aurora [1] at Brown and MIT, and Infopipes [42] at Georgia Tech have been working to formalize and implement the concepts for data-stream processing. Most of these efforts have commonly assumed an on-line warehousing model where all source streams are routed to a central site where they are processed. There have also been some preliminary proposals that extend the single-site model to multi-site, distributed models and environments [5, 62]. Our work is also a step in this general direction. Of particular mention is the work by Madden et al. [53] that demonstrates the advantage of in-network data-aggregation in a wireless multi-hop sensor network.

Distributed query optimization deals with site selection for the various operators and has been explored in great detail in the context of distributed and federated databases [27, 41]. However, these systems do not deal with streaming queries over streaming data, which presents new challenges, especially in dealing with resource

limitations.

2.5.2 Pub-Sub Middleware

Pub-sub middleware like IBM's Gryphon [65], ECho [25], ARMADA [2] and more recently Hermes [59] have well established themselves as messaging middleware. Such systems address issues like determining who needs what data, building scalable messaging systems and simplifying the development of messaging applications. We believe that our work is the necessary next step that utilizes the middleware to provide high-level programming constructs to describe resource-aware and 'useful' data-flows.

2.5.3 Network Partitioning & Overlay Networks

Distribution and allocation of tasks has been a long studied topic in distributed environments. Architectural initiatives tailored for large-scale applications include SkyServer [66], enterprise management solutions [30] and grid computing efforts [6]. These applications perform task allocation to servers much in the same way as we recursively map operators to nodes. However, a high-level construct for describing the data-flow and run-time re-assignment of operators based on an application-based utility distinguishes our infrastructure.

Overlay networks [64, 79] focus on addressing scalability and fault tolerance issues that arise in large-scale content dissemination. The intermediate routers in overlay network perform certain operations that can be viewed as in-network data-aggregation but are severely restricted in their functionality. The advantages of using our infrastructure are two-fold; first its ability to deploy operators at any node in the network, and second is the ease with which these operators can be expressed. There has also been some work on resource-aware overlays [80], which is similar to resource-aware reconfiguration of the stream overlay in our infrastructure. In our case reconfiguration is very closely associated with the application level data requirements.

2.6 Summary

The operational information system (OIS) is one of the more complex constituents of an enterprise's IT infrastructure. To address this complexity, our research has developed Dynamic Data Overlays Infrastructure, which provide a highly scalable solution for addressing the management issues that arise in an OIS. In particular, the infrastructure makes use of in-network data aggregation to distribute the processing and reduce the communication overhead in large-scale distributed data stream management. One of the important features of our approach is its ability to efficiently and scalably deploy data-flows across the network. The run-time reconfiguration of the deployed flow graph in response to change in operating conditions and support for high-level language constructs to describe data-flows are other distinguishing features of our approach. Experimental results show that our distributed data-flow graph deployment algorithm is able to achieve near-optimal deployment while ensuring the scalability of the solution approach. Similarly, the runtime reconfigurations are able to maintain the operational optimality of the deployed data-flows even when the operating conditions change with time.

Our interaction with the developers and managers at Delta Air Lines made us realize that the academic optima, i.e., optimizing for delay or bandwidth usage, may sometimes be orthogonal to what may be optimal for the enterprise. Any enterprise system that utilizes the underlying computing resources and incurs a corresponding cost also produces a certain benefit for the enterprise. The net utility of running an enterprise system can therefore be calculated as the difference between the benefit and the cost. The optimizations which revolve around using such formulations of utility are described in the following chapter.

CHAPTER III

*i*FLOW: UTILITY-DRIVEN MIDDLEWARE

3.1 Introduction

We consider enterprise-wide information flows that are responsible for acquiring, processing and delivering operational information across the business units. Middleware that enables such aggregation of data-streams must not only support scalable and efficient self-management to deal with changes in the operating conditions, but should also have an embedded sense of value of the data to appreciate the critical nature of some updates. This chapter describes *i*FLOW, an autonomic middleware for implementing enterprise-wide information in a utility-aware self-managing way. *i*FLOW offers a clean abstraction of an information flow and is therefore capable of implementing several other messaging models including the Enterprise information flows. *i*FLOW, in essence, is a utility-aware self-managing implementation of a general information flow abstraction. We describe the design and implementation of *i*FLOW, and describe case studies of implementing different messaging models as self-managing systems.

Distributed information-intensive applications range from emerging systems like continual queries [42], to remote collaboration [10] and scientific visualization [73], to the enterprise information systems used by large corporations [22]. At present, these applications utilize diverse messaging frameworks, where each such information flow framework implements basic primitives to acquire, process, and disseminate information across the underlying distributed system. Enhancements implemented over these primitives typically address the needs of specific application domains. Examples include scalability support like dynamic event routing in publish-subscribe systems, the

explicit SQL-like descriptions of information flows supporting continual queries and the tunable-operators found in middleware for scientific collaboration.

The applications described above, though dissimilar in their applications and implementation domains, fundamentally implement the same abstraction: distributed information flows. In particular, we can represent the information flows managed by each model using an information flow graph, consisting of descriptions of sources, sinks, flow-operators, edges and a utility-function. Sources represent the information producers, sinks are the consumers and flow-operators transform and combine information flows and they together constitute the vertices of the flow-graph. The edges represent information flows between such vertices. The utility-function measures the utility of an information flow as a function of attributes like bandwidth consumed, user-priority, etc. and acts as a vehicle for encoding user and system preferences for the purpose of flow self-management.

The advantage of reducing each of these models to a common abstraction is that one can embed autonomic features into the middleware implementing this abstraction. The information flow models then built on top of this middleware can directly benefit from the embedded autonomic features. For example, one can use the middleware implementing the flow graph abstraction to implement a pub/sub by modeling subscriptions [65] and event dissemination [25] as a flow graph, and then the autonomic flow graph can self-manage to ensure high-performance (which corresponds to high-utility) pub/sub. Alternatively, SQL-like information flows implementing continuous queries can also be realized using flow-graphs [46]. In such a scenario, the flow-operators would take the responsibility for 'join' or 'select' operations and the utility-function (probably corresponding to low-bandwidth usage) could be used to drive the self-management of the information flow. This chapter describes the design and implementation of *i*FLOW, a self-managing middleware substrate for building diverse messaging frameworks. *i*FLOW implements a set of core autonomic features

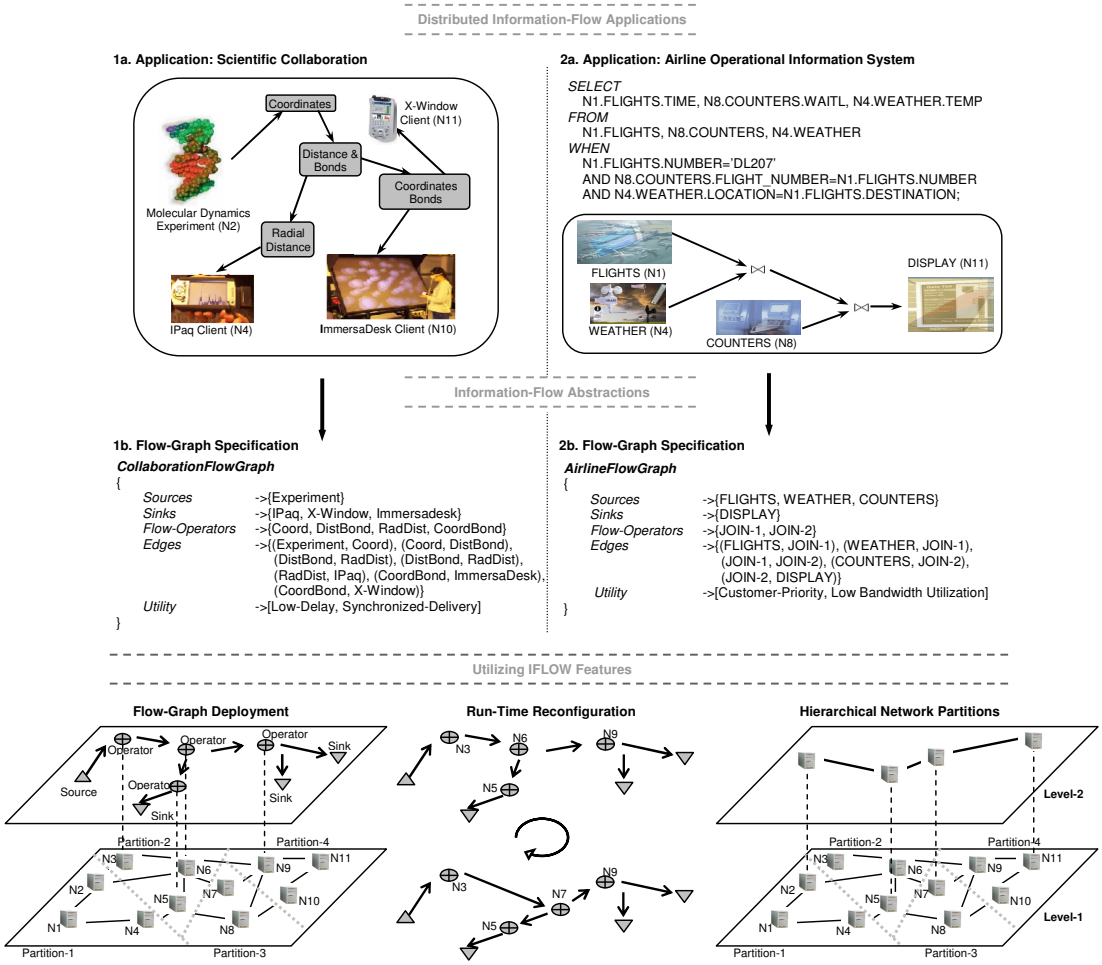

Figure 9: Implementation of different information flow models using *i*FLOW

that can benefit whatever information flow system is built on top of it. The utility-driven nature of the *i*FLOW middleware allows for creation of application-specific utility functions that govern both the initial deployment of information flows and their runtime regulation, thereby enabling the creation of application- or domain-specific self-regulation functionality.

3.1.1 Examples

Figure 9 depicts two examples of applications that can benefit from the generality of the *i*FLOW model and infrastructure - real-time scientific collaboration and an airline's operational information system.

The left portion of Figure 9 shows a collaborative real-time visualization application using a many-to-many information flow, with data originating in a parallel molecular dynamics (MD) simulation, passing through operators that transform and annotate the data, and ultimately flowing to a variety of clients. This real-time data visualization requires synchronized and timely delivery of large data volumes to collaborators. There may also exist some quality/performance trade-offs, as end users may prefer consistent frame rates to higher data resolution (or vice versa).

The right-hand information flow in Figure 9 represents elements of an enterprise information system (EIS) providing support for the daily operations of a company like Delta Air Lines (see [22] for a description of our earlier work with this company). Here, SQL-like operations translate into a deployed flow-graph with sources, sinks and operators. Such an OIS imposes the burden of high event rates on underlying resources, which must be efficiently utilized to deliver high utility to the enterprise. In such systems, business preferences like a 'high-priority' customer may need to be translated to the preferential routing of such updates through the system.

The remainder of this chapter will describe in more detail how *i*FLOW supports utility-driven self-management in applications like these and others. The main benefits of the *i*FLOW architecture for these applications are (1) autonomic features for self-configuring and continually optimizing information flows, and (2) the ease of implementing such information flows using declarative flow graphs and user/business utility specifications.

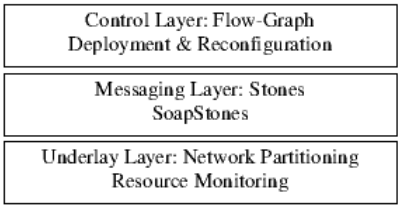

Figure 10: *i*FLOW software architecture

3.2 Software Architecture

In this section, we describe how the components of *i*FLOW are integrated to implement the abstraction of autonomic information flows. The architecture of *i*FLOW is shown in Figure 10 and we now describe each layer.

3.2.1 Control Layer

The Control Layer offers the abstraction of an Information Flow-Graph to applications. It is responsible for mapping a specified flow-graph onto some known underlay. Deployment is based on the resource information supplied by the underlay layer and a function for evaluating deployment utility. The application can specify a unique utility-function local to a flow-graph, or a global utility formulation can be inherited from the underlay layer. The control layer also handles the reconfigurations to maintain high utility for the deployed information flow. We now describe in detail the abstraction and functionality offered by this layer.

3.2.1.1 Information Flow-Graph

The information flow-graph is a collection of vertices, edges, and a utility-function, where vertices can be sources, sinks or flow-operators:

- A *source vertex* has a static association with a network node and has an associated data stream-rate. A source vertex can be have one or more outgoing edges.

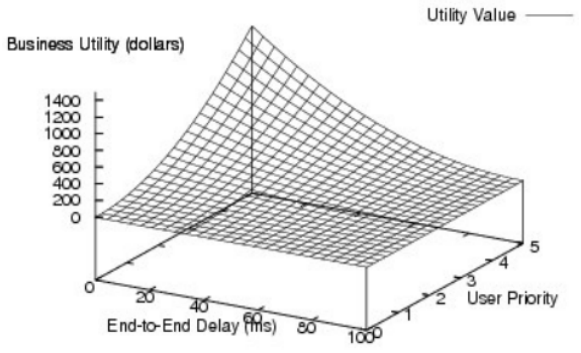

Figure 11: An example utility calculation model

- A *sink vertex* also has a static association with a network node. A sink vertex can have at most one incoming edge.

- An *operator vertex* can be associated with potentially any network node, and this association can change at runtime as the control layer reconfigures the flow-graph's deployment. Each operator is associated with a data resolution-factor (which is the ratio of the average stream output-rate to the average stream input-rate), an average execution-time, and the actual operator code (in E-Code [13]). An operator vertex can be associated with multiple incoming and outgoing edges.

*i*FLOW allows operators to be "pinned", if necessary, to specific network nodes (for example, if the node has specialized hardware.) Also, the internal control logic and parameters of operators can be remotely modified at runtime.

The utility of a flow-graph is calculated using the supplied utility-function and is based on both application-level (e.g., user-priority) and system-level (e.g., delay) attributes. The function can be used to calculate the net-utility of a flow-graph mapping by subtracting the cost it imposes on the infrastructure from the utility-value. For example, consider a utility formulation expressed as:

$$utility = (k - delay)^2 \times priority \tag{2}$$

The formulation (shown in Figure 11) depicts a system where end-to-end delay and the user-priority determine the utility of the system. The utility model in this scenario can be stated as 'High Priority users are more important for the business' and 'Less end-to-end delay is better for the business'.

3.2.1.2 Flow-Graph Construction & Mapping

We observe that there are two distinct classes of information flows: basic and semantic.

Basic information flows arise in applications in which only the sources and sinks are known, and the structure of the data flow-graph is not specified. For example, in the pub-sub model, there exists no semantic requirement on the flow-graph. *i*FLOW can establish whichever edges and merge/split operators may be necessary, between publishers and subscribers. To accommodate this class of 'basic' information flows, we have developed a novel graph construction algorithm, termed InfoPath. InfoPath uses the resource information available at the underlay layer to both construct an efficient graph and map the graph to suitable physical network nodes.

Semantic information flows are flows in which the data flow-graphs are completely specified, or at least have semantic requirements on the ordering and relationship between operators. For example, SQL-like continual queries over data streams specify a particular set of operations based on the rules of relational algebra. Similarly, in a remote collaboration application, application-specific operators must often be executed in a particular order to preserve the data semantics embedded in scientific work-flows. Here, *i*FLOW takes the specified flow-graph and maps it to physical network nodes using the PathMap mapping algorithm. This algorithm utilizes resource information at the underlay layer to map an existing data flow-graph to the network in the most efficient way. The InfoPath and PathMap algorithms are described in detail in Section 3.3.

3.2.1.3 Reconfiguration

After the initial efficient deployment has been produced by PathMap or InfoPath, conditions may change, requiring the deployment to be reconfigured. *i*FLOW maintains a collection of configuration information, called the IFGRepository that can be used by the control layer when reconfiguration is necessary. As shown in Section 3.3, the underlay layer uses network-awareness to cluster physical nodes, and the IFGRepository is actually implemented as a set of repositories, one per underlay cluster. This allows the control layer to perform local reconfigurations using local information whenever possible. Global information is accessed only when absolutely necessary. Thus, reconfiguration is usually a low-overhead process.

The *i*FLOW framework provides an interface for implementing new reconfiguration policies based on the needs of the application. Our current implementation includes two reconfiguration policies: the Delta Threshold Approach and the Constraint Violation Approach. Both approaches take advantage of the IFGRepository and the resource information provided by the underlay layer to monitor the changes in the utility of a graph deployment. The Delta Threshold Approach and the Constraint Violation Approach are described in greater detail in Section 3.3.4.

3.2.2 Messaging Layer

The messaging layer of *i*FLOW is composed of communicating objects, called Stones, which are linked to create data paths. Stones are lightweight entities that roughly correspond to processing points in data-flow diagrams. In particular, Stones are responsible both for the routing and dissemination of data, and for processing it as it flows. Because Stones can be created and destroyed, and the behavior of Stones can be changed at runtime, they provide a useful infrastructure for dynamically reconfiguring information flows as needed to continually optimize flow utility.

Application data enters the system via an explicit submission to a Stone, but

thereafter it travels from Stone to Stone, sometimes crossing network links, until it reaches its destination. The actual communication between Stones in different processes is handled by the Connection Manager [25], a transport mechanism for heterogeneous systems, which uses a portable binary data format for communication and supports the dynamic configuration of network transports through the use of attributes.

Each Stone is associated with a set of actions and queues. Actions are application-specified handlers that operate on the messages handled by a Stone. Examples of actions include filtering messages depending on their contents, converting the type and format of message data, transforming data by applying an application-supplied calculation to it, splitting streams for forwarding to multiple downstream stones, and passing messages up to the application. Thus, Stones provide the interface to sources and sinks for the information flow, as well as the platform for implementing data flow operators. The handler functions are specified in E-Code, a portable subset of the C language, and can be downloaded into the Stone at runtime. Dynamic code generation [28] is then used to install and execute the handlers. Not only does this process facilitate dynamic reconfiguration, but it also permits the handlers to be run on heterogeneous platforms. Queues associated with Stones: (1) synchronize incoming events to a Stone that operates on messages coming from multiple Stones and, (2) temporarily hold messages when necessary during reconfiguration.

To enable remote invocation of stone primitives we have implemented a SOAP enabled extension for stones termed SoapStone.

3.2.3 Underlay Layer

The Underlay Layer is primarily responsible for maintaining a hierarchy of physical nodes in order to cluster nodes that are 'close' in the network sense. This hierarchy (see Figure 12) is adapted from our work on Dynamic Data Overlays Infrastructure

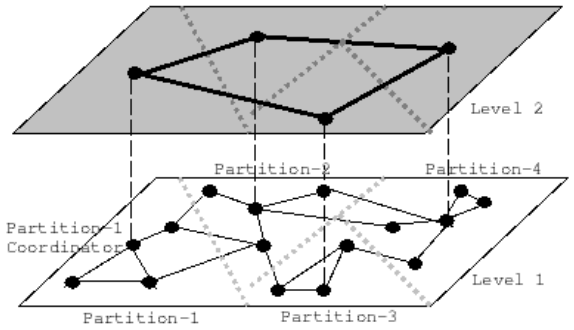

Figure 12: Hierarchical network partitioning

and is summarized here for completeness. Each cluster elects a coordinator which serves as the cluster's representative and supervises the self-management of the cluster. Coordinators can themselves be grouped into a second layer of clusters, each of which similarly elects a coordinator. Repeating this process, we can form a hierarchy of clusters. Information flows are deployed by starting at the top of the hierarchy and recursively partitioning the flow graph until we reach the lowest layer (as described in the next section.) The underlay layer is also responsible for handling node Join and Departure requests. Finally, the cluster coordinators monitor resource availability in the cluster and report resource events via the Proactive Directory Service [13].

3.3 Algorithms

We now describe in detail the various algorithms built into our framework. First, we present a formal model of our framework, and then discuss each algorithm.

3.3.1 Framework

The information flow-graph is represented as $G(V_g, E_g, U)$ with each vertex in V_g representing a source-node, a sink-node or an operator: $V_g = V_g^{sources} \cup V_g^{sinks} \cup V_g^{operators}$. The edges E_g in the information flow-graph represent the flow of information, and may span multiple intermediate edges and nodes in the underlying network. Finally, the utility-function U contains a formulation based on application-level and

resource-level attributes to calculate the utility of any edge in the flow-graph, given the system conditions. The utility of the flow-graph U_G, can then be calculated as: $U_G = \sum_{e \in E_G} U(e)$. The underlying network is represented as a network graph $N(V_n, E_n)$ where vertices $v_{nj} \in V_n$ represent the actual physical nodes and the network connections between the nodes are represented by the edges $e_{nj} \in E_n$. We further associate each edge e_{ni} with a delay d_{ni} and available-bandwidth b_{ni} that are derived from the corresponding network link.

3.3.2 InfoPath

Problem: The InfoPath problem is to construct an appropriate path between the sources and sinks of the information flow-graph when no such path has been defined, and when all data emanating from the sources must be delivered to the sinks. The algorithm makes use of two standard operators, which merge or split the information flows. It instantiates a number of appropriate edges to maximize the net-utility (i.e., the difference of utility obtained from the deployed graph minus the cost it imposes on the system) derived by connecting sources to the sinks. More formally, we want to instantiate a number of merge/split operators $v_{gj} \in V_g^{operators}$ assigned to some $v_{ni} \in V_n$, and edges $e_{gj} \in E_g$ such that the resulting graph is a maximum net-utility graph for data dissemination.

Solution: A naive solution to routing information from sources to sinks would establish one-to-one connections. However, this approach can be very expensive. Instead, we exploit the hierarchical organization of nodes to construct a flow-graph that acts as an application-level multicast tree to efficiently disseminate updates. Towards this end, the InfoPath algorithm proceeds in the following manner. First, a set of sources and sinks ($V_g^{sources} \cup V_g^{sinks}$)is submitted as input to the top-level (say level t) coordinator along with a general utility formulation U. The sources and the sinks trickle down into their partitions at Level 1, the level at which the physical

nodes reside, presenting each partition with a subset of initial sources and sinks. The partition coordinator then finds the maximum net-utility path for connecting the subset of nodes using one-to-one connections. This is based on the observation that it is not too costly for InfoPath to establish one-to-one connections between the sources and the sinks within a single partition. Since the hierarchical partitioning algorithm clusters nodes according to network 'closeness', within a partition the data transmission costs are low. Then, the algorithm further condenses all the sources in a partition into a single virtual source using a merge operator, and similarly creates a single virtual sink for all the partition's sinks using a split operator. The virtual source and sink resulting from each partition form the sources and sinks managed by the coordinator at the next level of the hierarchy. At that level, we again connect virtual sources and sinks using one-to-one connections, and this process proceeds recursively until level t - 1. The InfoPath algorithm is used in the implementation of the Pub-Sub model described in Section 3.4.3.

3.3.3 PathMap

Problem: Given a flow-graph G, we want to produce a mapping M, which assigns each $v_{gj} \in V_g^{operators}$ to a $v_{ni} \in V_n$. Thus, M implies a corresponding mapping of edges in G to edges in N, such that each edge e_{gj-k} between vertices v_{gj} and v_{gk} is mapped to the network edges along the maximum net-utility path between the network nodes that v_{gj} and v_gk are assigned to. We define $netUtility(M_{j-k})$ as the difference between the utility obtained from deploying the flow-graph edge e_{gj-k} and the cost imposed by the deployment M_{j-k} of e_{gj-k} across v_{nj} and v_{nk}.

$$netUtility(M_{j-k}) = utility(e_{gj-k}) - \sum_{e_n \in M(e_{gj-k})} cost(e_n) \tag{3}$$

Using the above formulation, we can define the net-utility of flow-graph G, produced by a mapping M as:

$$netUtility(M) = \sum_{e_g \in E_g} (utility(e_g) - \sum_{e_n \in M(e_g)} cost(e_n)) \quad (4)$$

For example, consider a cost function that is based on the amount of data transferred (i.e. bandwidth consumed). If e_{gk} is determined by vertices v_{gi} and v_{gj}, which in turn are assigned to vertices v_{ni} and v_{nj} of the network graph N, then the cost corresponding to edge e_g is cost of transferring data along the lowest-cost path between the vertices v_{ni} and v_{nj}. Our goal is to construct a mapping M of vertices that implies the maximum net-utility mapping between the edges $E_g \in G$ to edges $E_n \in N$. For example, the costs of transferring data along different physical links differ. However, by paying the cost, certain guarantees (such as average delay experienced) are achievable. Rather than blindly minimizing the cost of deploying a flow-graph, we try to maximize the net-utility achievable.

Solution: The PathMap algorithm works as follows. The given information flow-graph $G(V_g, E_g, U)$ is submitted as input to the top-level (say level t) coordinator. We construct a set of possible node assignments at level t by exhaustively mapping all the vertices $V_g^{operators} \in V_g$ to the nodes at this level. The hierarchical structure and limited size of partitions ensures that there are few nodes at this level, minimizing the expense of the exhaustive mapping. The net-utility for each assignment is calculated and the assignment with maximum net-utility is chosen. This partitions the graph G into a number of sub-graphs each allocated to a node at level t and therefore to a corresponding partition at level t - 1. The sub-graphs are then again deployed in a similar manner at level t - 1. This process continues till we reach level 1, which is the level at which all the physical nodes reside, and operators are assigned to actual physical nodes. Because the graph is recursively deployed using the hierarchical partitioning structure, edges are kept inside a partition whenever possible,

and whenever an edge must cross partitions a maximum net-utility path between the two partitions is chosen. PathMap is used in our implementation of collaborative visualization (Section 3.4.1) and operational information systems (Section 3.4.2).

3.3.4 Reconfigurations

The initial deployment will have high business utility. However, when conditions change, utility can drop, and it may become necessary to reconfigure the deployment. The overlay reconfiguration process takes advantage of two important features of our infrastructure: (1) that the nodes reside in clusters and (2) that only intra-cluster maximum utility analysis is required. These features allow us to limit the reconfiguration to within the cluster boundaries for local fluctuations, which in turn makes reconfiguration a low-overhead process. An overlay can be reconfigured in response to a variety of events, which are reported to the first-level cluster-coordinators by the PDS. These events include change in network delays, change in available bandwidth, change in data-operator behavior (we call this operator profiling), available processing capability, etc. There are also some business specific events that may trigger reconfiguration. One example is time-of-day, if the system has different observed traffic behaviors during different times of the day. Consider region specific websites that provide local news, weather, etc. These sites tend to have high traffic during morning hours, and the system goal is to maintain high-availability during this duration. Another example of a business event is the arrival of high-priority customers.

Reconfigurations can be expensive and temporarily disrupt the operation of the system. Therefore, it is impractical to respond to all such events reported by the PDS and the business. Instead, our system limits the number of reconfigurations it performs. We examine two approaches to limiting reconfigurations. One approach is to set certain thresholds that should be reached before a reconfiguration occurs. For example, a cluster-coordinator may recalculate the maximum utility paths and

redeploy the assigned sub-graphs when more than half the edges in the cluster have reported change in utility value. Another example is to trigger a reconfiguration when the utility from a deployed graph falls by more than a threshold when compared with the best achievable utility. Another approach to limiting reconfigurations is to specify that a reconfiguration should only occur when certain constraints have been violated. For example, we may specify an upper bound on end-to-end delay, and trigger a reconfiguration when the total end-to-end delay violates this upper bound. The two approaches are examined in the following subsections.

3.3.4.1 Delta Threshold Approach

In this approach, when we deploy a graph we specify the maximum tolerable negative deviation from the best achievable utility. Since in our system the graph is partitioned into sub-graphs, we can either aggregate deviations from the sub-coordinators to manage threshold at a central planner, or we can distribute smaller thresholds to sub-coordinators such that the sum of smaller thresholds is equal to the overall threshold and each sub-coordinator manages the assigned threshold. We chose to implement the latter approach, and experimental results for this approach are presented in Section 3.5.

3.3.4.2 Constraint Violation Approach

This approach triggers reconfiguration when a constraint on a system parameter such as end-to-end delay or available-bandwidth is violated. The rationale behind this approach is that it may be easier to check for the violation of a constraint than to maintain a threshold against the optimal business utility. We again divide and distribute the constraint value to sub-coordinators, such that the constraint is not violated as long as all the sub-coordinators manage to satisfy their local constraint.

Note that the operators in our middleware can have state associated with them and therefore a reconfiguration can be lossy or lossless. Our middleware provides the

capability to choose the type of reconfiguration and such a choice affects the cost associated with the reconfiguration process. We make use of existing checkpointing mechanisms to implement lossless reconfigurations.

3.4 Case Studies

3.4.1 Collaborative Visualization

Our first application is the collaborative visualization of a molecular dynamics simulation, which is of interest to computational scientists in a wide variety of fields, from pure science (physics and chemistry) to applied engineering (mechanical and aerospace engineering). The visualization application is geared to deliver the events-of-interest to participating collaborators, formatted to suit the rendering capabilities at their ends. More details about this application are available in [73]. Application development for such a collaboration using *i*FLOW involves identifying sources (simulation sites), sinks (scientists with rendering equipments), operators (filters for events-of-interest, formatting or downsizing of data to suit end-user capabilities), edges (representing the data-flow) and a utility function that encodes the requirement for information-freshness (low-delay), low bandwidth-utilization and synchronized-delivery (similar delays along various information flow-paths). Then, an Information Flow-Graph can be constructed using *i*FLOW primitives and submitted to the underlying infrastructure for actual deployment using PathMap. Deployment of an information flow requires the existence of a boot-strapped middleware. *i*FLOW's underlay provides primitives that automatically initialize and organize the underlying nodes that desire to participate in the middleware. Once the participating nodes are up and running, the job of deploying the collaboration flow-graph is simple - a call to the PathMap algorithm at any node in the infrastructure with the constructed flow-graph. Once the flow-graph has been mapped onto the infrastructure nodes, reconfiguration is taken care of by *i*FLOW in accordance with the supplied

utility-function.

3.4.2 Operational Information System

This application is inspired by the use-case and scenarios provided to us by our long-standing collaborator, Delta Air Lines. An operational information system, as described in Section 3.1.1, provides continuous support for an organization's daily operations. We implement an information flow motivated by the requirement to feed overhead displays at airports with up-to-date information. The overhead displays periodically update the weather at the 'destination' location and switch over to seating information for the aircraft at the boarding gate. Other information displayed on such monitors includes the names of wait-listed passengers, the current status of flights, etc. We deploy a flow-graph with two operators, one for selecting the weather information (which originates from a weather station) based on flight information, and other for combining the appropriate flight data (which originates from a central location like Delta's TPF facility) with periodic updates from airline counters that decide the wait-list order, etc. Thus, the three sources can be identified as - the weather information source, the flight information source, and the passenger information source. They are then combined using the operators to be delivered to the sink - the overhead display. A detailed discussion of the utility-driven deployment of such a flow-graph can be found in [49].

3.4.3 Pub-Sub Implementation

We have also implemented a simple publish-subscribe messaging model using the *i*FLOW framework. In this implementation, messages are sent via publishers into a pubsub channel, which may have zero or more subscribers. The subscribers may be on the same machine as the publisher or anywhere else in the network; however, their locations are immaterial to the publisher. Each subscriber receives only the messages sent to the channel to which it is subscribed. The development of such a messaging

model requires abstracting the publishers and subscribers as sources and sinks (respectively) in the information flow-graph; it also requires an application-dependent utility formulation that can drive deployment. The InfoPath algorithm, described earlier, provides the capability to establish an efficient information flow path between a given set of sources and sinks for message dissemination. Each pub-sub channel is assigned a globally unique identifier, essentially identifying the information flow-graph, or a part of it, stored in the IFGRepository of coordinators across the network. Once the information flow has been initially deployed on a boot-strapped infrastructure, and an identifier has been obtained for the deployment, the task of adding/removing participants can be handled as follows: Adding a publisher requires identifying the cluster to which the publisher belongs, setting up one-to-one connections with the sinks in the cluster, and finally establishing a connection with the merge operator (virtual source for next level in the hierarchy) corresponding to this flow-graph. Adding a subscriber requires identifying the cluster to which the subscriber belongs, establishing intra-cluster one-to-one connections with the publishers, and subscribing to the virtual sink for out-of-cluster updates. The task of removing participants can be implemented trivially at the cluster level and similar to additions requires no global exchange of messages.

3.5 Experiments

The purpose of experiments is to evaluate the performance of our *i*FLOW middleware. Microbenchmarks examine specific system features. Results show that our system is effective at self-configuration and self-optimization of data-flow graphs for distributed processing of streaming data.

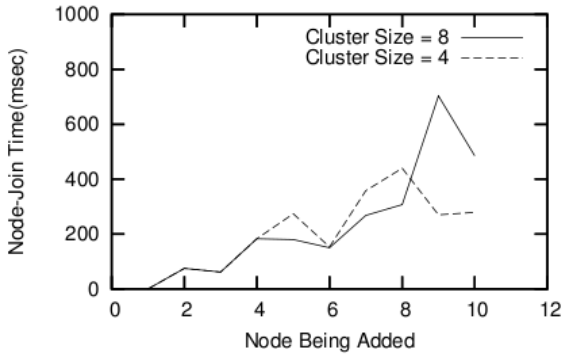

Figure 13: Time to add N^{th} node to the underlay

3.5.1 Microbenchmarks

3.5.1.1 Underlay Layer: Scalability

The hierarchical partitioning of nodes at the underlay layer provides several performance benefits. First, the underlay simplifies the task of adding or removing network nodes. Figure 13 shows the time taken to add the Nth physical node to the middlewares underlay layer for cluster sizes 4 and 8. These results indicate that node-join times scale well with increasing number of nodes. Any node that joins the underlay finds its appropriate cluster, an O(logN) operation, followed by a constant time operation to determine intra-cluster attributes, which in-turn will depend on the maximum cluster size. The underlay layer is also critical to the performance of the flow-deployment algorithms implemented at the control layer (evaluated in Section 3.5.2.2), and the add/remove facility for pub-sub participants (evaluated in Section 3.5.2.3)

3.5.1.2 Messaging Layer: Stone Performance

We also measured the performance of Stone operations, since efficient utility-driven data dissemination and data-flow reconfiguration rely on fast Stones. The following micro-benchmarks are measured using a 2.8 GHz Xeon quad processor with 2MB cache, running Linux 2.4.20 smp as a server. The client machine used is a 2.0 GHz

Table 1: Stone: send and receive costs

Message Size KB	100	10	1
Receiver Cost μsec	17.4	14.3	6.8
Sender Cost μsec	9.3	5.4	5.3

Table 2: Stone microbenchmarks in μsec

Operation	Time
Stone Create	0.89
Stone Delete	0.03
Source Stone Create	1663.94
Sink Stone Create	8.31

Xeon quad processor, running Linux 2.6.10 smp. Both the machines are connected by single-hop 100Mbps ethernet.

Send/Receive Costs: A stone's most significant performance feature is its use of the native data format on the sender side, coupled with dynamically generated unmarshalling code at the receiver to reduce the send/receive costs. As shown in Table 1, both the send side cost and receive side cost for these operations are small compared to the typical round trip delays experienced in local area networks (about 0.1-0.3ms with a Cisco Catalyst 6500 series switch) and negligible for typical wide area round trip delays (50ms-100ms).

Throughput Comparison against MPICH: We also compare the throughput achieved for different message sizes using Stones to that of raw sockets and MPICH. The results (not shown) indicate that achieved throughput values closely follow the raw socket throughput, and are slightly better than the values achieved using MPICH. This is very encouraging for *i*FLOW applications that target the high performance domain.

Stones Instantiation Overheads: The deployment of a flow-graph at the overlay layer consists of creating source, sink or filter Stones and associating suitable action

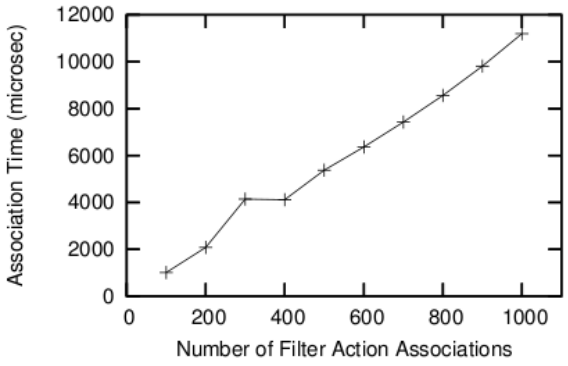

Figure 14: Time to associate filter actions

routines to the Stones. Table 2 show the small delays required for local Stone actions, making them suitable for supporting reconfigurations that require frequent Stone creation and deletions. The comparatively high cost for source Stone creation is due to format registration but this is done only once and hence is not a problem.

Multiple Filter Association Times: To facilitate operations to be performed on data flowing through the Stones, we would need to associate filter actions to the corresponding Stones. Figure 14 shows the time involved in associating filter action to a Stone. All associations are performed on the same Stone, but the destination Stones were changes. The graph shows a linear trend, indicating that filter Stones are suitable for large- as well as small-scale deployments.

3.5.2 Control Layer: Algorithms

3.5.2.1 Experimental Setup

The GT-ITM internetwork topology generator [76] was used to generate a sample Internet topology for evaluating our self-configuration algorithm. This topology represents a distributed OIS scattered across several locations. Specifically, we use the transit-stub topology for the ns-2 simulation by including one transit domain that resembles the backbone Internet and four stub domains that connect to transit nodes

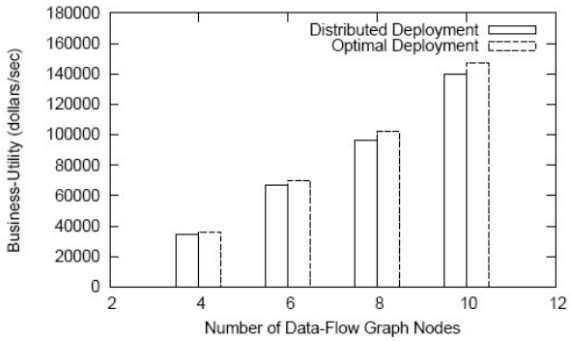

Figure 15: Comparison of the utility achieved using centralized versus PathMap approach

via gateway nodes in the sub domains. Each stub domain has 32 nodes and the number of total transit nodes is 128. Links inside a stub domain are 100Mbps. Links connecting stub and transit domains, and links inside a transit domain are 622Mbps, resembling OC-12 lines. The traffic inside the topology was composed of 900 CBR connections between sub domain nodes generated by cmu-scen-gen [18]. The simulation was carried out for 1800 seconds and snapshots capturing end-to-end delay between directly connected nodes were taken every 5 seconds. These are then used as inputs for our distributed deployment algorithm. This is the experimental setup used for the following evaluations unless specified otherwise.

3.5.2.2 PathMap Evaluation

Our first experiment focused on comparing the business-utility of a deployed data-flow graph using the centralized model as opposed to the partitioning based PathMap approach. Since the centralized approach assumes that a single node tracks utility-determining system parameters for all other nodes, it produces the optimal deployment solution. However, the deployment time taken by the centralized approach increases exponentially with the number of nodes in the network. Figure 15 shows that although the partitioning-based approach is not optimal, the business-utility of the deployed flow graph is not much worse than the deployment in the centralized

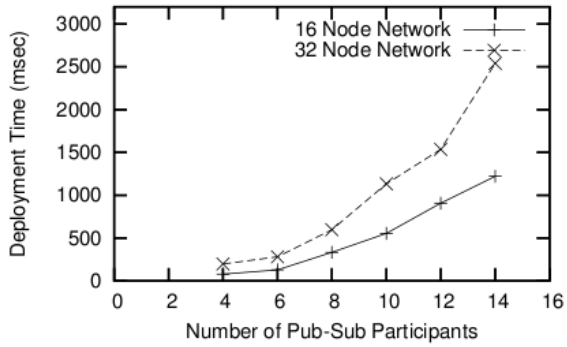

Figure 16: Deployment times using InfoPath for two different network sizes

approach, and is thus suitable for most scenarios.

3.5.2.3 InfoPath Evaluation

The next set of experiments uses an Emulab setup with 16 and 32 node (Intel XEON, 2.8 GHz, 512MB RAM, Red-Hat Linux 9) topologies, generated with GT-ITM. Links are 100Mbps and the inter-node delays are set between 1msec and 6msec. The first experiment measures the time taken by our implementation of pub-sub (described in Section 3.4.3) to establish the links between the pub-sub participants. Our implementation uses the InfoPath algorithm that tries to establish a near-optimal update dissemination path between the participants using the information provided to it by the underlay. The time taken to deploy a pub-sub with varying number of participants on 16 and 32 node networks is shown in Figure 16. The low deployment times can be attributed to the bound on the partition size (e.g., 8), and to the fact that link establishment between sub-groups of participants residing in different partitions occurs in parallel.

We also measure the average time taken to add or remove a participant from a 10 node pub/sub graph. For a 16 node network, these times were 67.9 and 48.8 ms respectively, and for a 32 node network these times were 78.4 and 54.6 ms respectively. These operations are quick because they are limited to a single underlay partition.

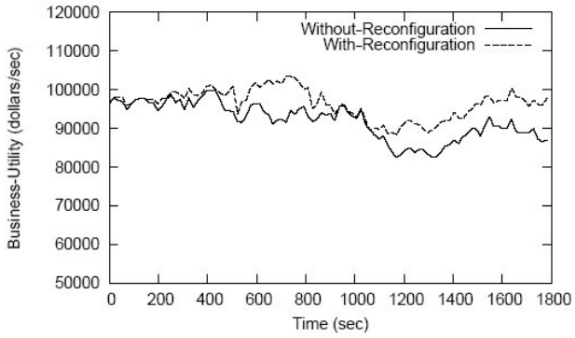

Figure 17: Utility variation for a data-flow graph with and without self-optimization

3.5.2.4 Reconfiguration

The next experiment is conducted to examine the effectiveness of dynamic reconfiguration in providing an efficient deployment. Figure 17 shows the variation of business-utility for a 10-node data-flow graph with changing network conditions, as simulated by introducing cross-traffic. The performance with reconfiguration is clearly better than without reconfiguration. Note that at some points, the utility of the flow-graph with reconfiguration becomes less than that of flow-graph without reconfiguration. This happens because the hierarchical utility calculation algorithm used in our approach calculates the graph utility that is an approximation of the actual business utility. In some cases the approximation is inaccurate, causing the reconfiguration to make a poor choice. However, these instances are rare, and when they do occur, the utility of the flow with reconfiguration is not much worse than the one without reconfiguration. Moreover, for most of the time reconfiguration produces a higher utility deployment.

We also conduct experiments to evaluate the performance of our two self-optimization approaches: the delta threshold approach, and the constraint violation approach. The change in business-utility of a 10-node data flow graph using the delta-threshold approach in the presence of network perturbations (the traffic was composed of 900 CBR connections between sub domain nodes generated by cmu-scen-gen) is shown in

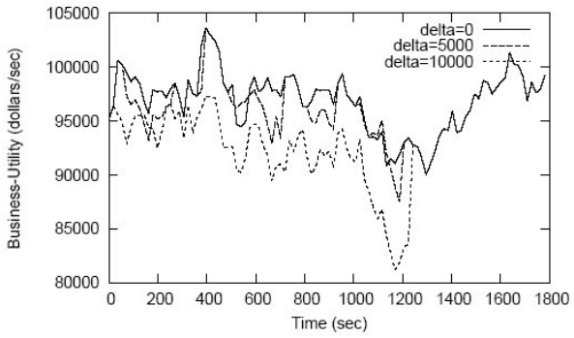

Figure 18: Utility variation using Delta-Threshold approach in presence of network perturbation

Figure 18. We notice that even for a sufficiently large value of threshold the achieved utility closely follows the maximum achievable utility. We also calculate the corresponding capital loss incurred due to sub-optimal deployment of the flow-graph using different thresholds. The results are shown in Table 3. The loss is calculated as the integral over time of the difference between the maximum achievable utility and the current utility for the deployed flow graph. The loss incurred increases exponentially as the threshold is increased. However, it is sufficiently low for a large number of values, and thus an appropriate threshold value can be used to trade-off utility for a lower number of reconfigurations.

Figure 19 shows the variation of business utility when the constraint-violation approach is used for self-optimization. In this experiment, we place an upper bound

Table 3: Loss & Reconfiguration using Delta-Threshold approach

Threshold (dollar/sec)	Reconfiguration Count	Capital Loss (dollars)
0	11	0
2500	7	9881
5000	6	250326
7500	4	876471
10000	1	5728104

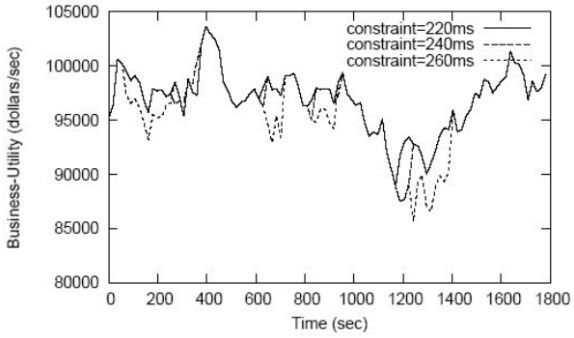

Figure 19: Utility variation using Constraint-Violation approach in presence of network perturbation

Table 4: Loss & Reconfiguration using Constraint-Violation approach

Constraint (msec)	Reconfiguration Count	Capital Loss (dollars)
220	10	0
230	7	112987
240	6	292509
250	6	976588
260	6	1860597

on the total end-to-end delay for the deployed data-flow graph, and trigger a reconfiguration when this bound is violated. This experiment is driven by the real world requirement for delaying reconfiguration until a constraint is violated, because in some scenarios it might be more important to maintain the configuration and satisfy minimal constraints rather than optimize for maximum utility. We can notice some resemblance in behavior between the delta-threshold approach and the constraint violation approach. This is because utility is a function of end-to-end delay for the deployed flow graph. However, managing the system by monitoring constraint violation is far easier than optimizing a general utility function. Self-optimization that is driven by change in utility value is more difficult than the one driven by constraint violation, because calculating maximum achievable utility requires knowledge of several system parameters and of the deployment ordering amongst various graphs for achieving maximum utility. The corresponding capital loss due to different constraint

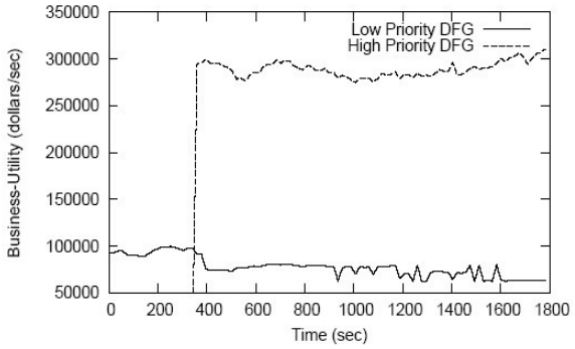

Figure 20: Effect of injecting high priority flow-graph

values and the reconfigurations are shown in Table 4.

3.5.2.5 Injecting Data-Flows

Next, we conduct an experiment to study the effect of deploying a high priority data-flow graph in a system with an existing deployed low priority data-flow graph. A 10-node data-flow graph with priority=1 is introduced at time t=0.0 sec and then at t=360.0 sec another 10-node data-flow graph with priority=3 is injected into the system. Figure 20 shows the resulting business utility over time. We see a noticeable decrease in the utility for the low-priority graph when the new flow-graph is injected. This is because the high priority graph is preferentially assigned system resources. Thus, the bandwidth and the minimum delay paths previously available to the low-priority graph may get assigned to the high-priority graph thereby reducing its net business utility.

3.5.2.6 Policy driven Self-Optimization

Next, we conduct an experiment to study the responsiveness of our middleware to changes in the business policy. The priority of a data-flow graph in our system is determined by the maximum user-priority amongst all users accessing that data-flow. We initiate 3 different 10-node data-flow graphs and simulate a number of users with 5 different priority levels in the presence of network perturbation. The priority of

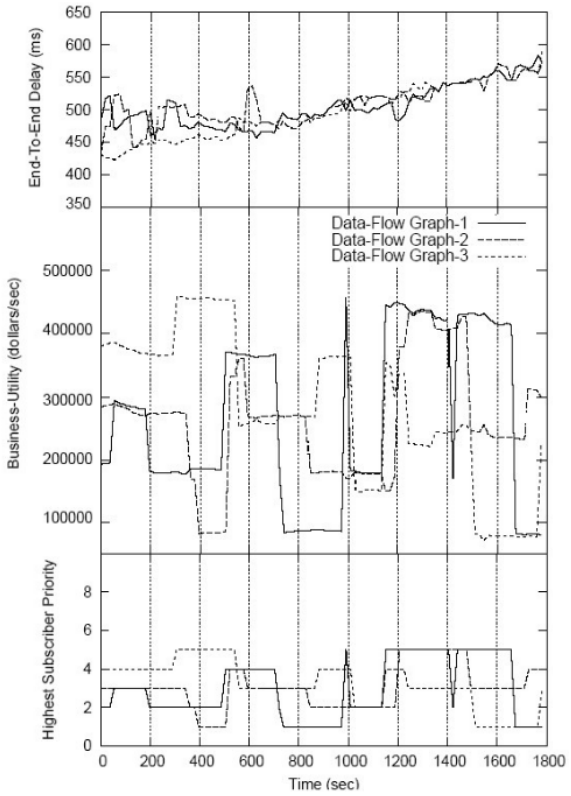

Figure 21: Effect of policy-driven self-optimization

the data-flow graphs is driven by the user accessing the data and changes as the users arrive and depart. The change in business-utility corresponding to fluctuating priority is shown in Figure 21. There is an almost instantaneous change in utility value when the priority of a particular graph changes. An interesting result shown in the figure is the corresponding change in delay for the graph as the priority of flow changes. We notice a decrease in delay for data-flows when there is a corresponding increase in graph priority. This result is interesting because the system autonomically assigns lower delay paths to flows with higher priority using the utility function.

3.5.3 Comparison with Delta's Middleware

Delta Technology Messaging Interface (DTMI) is a messaging middleware developed by Delta Technology (DT) to support demanding enterprise-class applications built using a service oriented architecture. Application services are deployed in server

Table 5: Time to process and propagate 10,000 events on *i*FLOW and DTMI (sec)

Event Size (bytes)	*i*FLOW	DTMI
256	9.29	26.01
1024	10.43	27.63
5120	17.18	29.03
10240	27.33	34.62

processes running on a cluster of heterogeneous machines. Services communicate by sending messages to each other, and high throughput results from using low-latency System V message queues for communication on the same host while socket communication (over TCP) is used for message passing across machine boundaries.

This evaluation is conducted to substantiate our claim that *i*FLOW performs competitively against an industrial strength middleware. We use Emulab [26] to create a 10 node (Intel XEON, 2.8 GHz, 512MB RAM, RedHat Linux 9) topology that was generated using GT-ITM. Links are 100Mbps and the inter-node delays are set between 1msec and 6msec. We then instantiate the flow-graph (three sources, two operators and a sink) described in Section 5.2 using both *i*FLOW and DTMI. Table 5 shows the time taken by each system to process 10,000 events of different sizes. *i*FLOW performs competitively with DTMI. In fact, DTMI is slower than *i*FLOW, but this is because DTMI provides more services, such as fault tolerance and sanity checks. Thus, we do not claim that *i*FLOW is truly faster, but our results provide evidence that *i*FLOW performs competitively against an industrial strength middleware.

3.6 Related Work

The development of the *i*FLOW middleware was a logical step towards enabling utility-driven self-management for the dynamic data overlays infrastructure. Besides being related to the work discussed in related work section of Chapter 2, the *i*FLOW middleware closely relates to the work being done as part of many self-management

architectures and with the work on autonomic computing and utility-functions.

3.6.1 Self-Managing Services, Architectures, and Infrastructures

Systems like Astrolabe [70] have the capability to self-configure, monitor and adapt a distributed hierarchy to manage evolving data-sources. *i*FLOW builds on these concepts by providing a general substrate for different kinds of flows, and by incorporating utility functions. Other examples of self-managing systems include the service recipes in Darwin [36], or the low cost service deployment in XenoServer [43]. It may be possible to extend *i*FLOW with these capabilities.

3.6.2 Autonomic Computing & Utility Functions

The tremendous increase in complexity of computing machinery across the globe and the resultant inability of administrators to deal with it, has initiated activities in academia and industry to make systems self-managing. A vision of autonomic computing as described in [40] is to design computing systems that can manage themselves given high-level objectives from administrators. Of the four aspects of self-management defined in the vision, *i*FLOW focuses on self-configuration, self-optimization and self-healing in stream management middleware. Our architectural approach is similar to earlier work in adaptive systems, captured for the autonomic domain in [34]. While the utility-driven self-optimization in our system is inspired by earlier work in the real-time and multimedia domains [44], the specific notions of utility used in our work mirror the work presented in [67] which uses utility functions for autonomic data-centers. Utility functions have been extensively used in the fields of economics [57] and artificial intelligence [61]. Autonomic self-optimization according to business objectives has been studied in [32] although the distributed and heterogeneous nature of resources makes our infrastructure more general.

3.7 Summary

This chapter presents the design, implementation, and experiences with *i*FLOW, a novel autonomic messaging middleware that can be used to develop distributed applications that involve the acquisition, processing and dissemination of updates. *i*FLOW models such an application as an Information Flow-Graph, thus encompassing the ability to express the requirements of several messaging applications. The approach is novel in its explicit consideration of business value during deployment and when configuration changes occur. The formalized notion of an information flow-graph proves equally useful for implementation of highly scalable deployment and reconfiguration algorithms at the control layer. The Stones and queues at the messaging layer serve as the building blocks for the information overlays across the network, efficiently assisted by the underlay layer, which provides resource-aware system partitions. Extensive experiments conducted using the *i*FLOW infrastructure demonstrate the highly scalable and utility-aware nature of the middleware and its ability to handle at runtime the changes in operating environment.

The utility formulations used in our research for enabling self-management are encapsulations of the system model in mathematical terms, but unfortunately, it is always not possible to devise a useful and realistic formulation of utility for actual large-scale systems. This was further corroborated by our interaction with our industry collaborators, who at most times, portrayed their systems as being highly unpredictable, let alone being able to come up with a utility formulation for the system. This motivated us to look at techniques that can be used to learn a probabilistic model of a system's behavior based on observed system data, which can then be used to enable self-management of such systems. The *i*Manage framework, described in Chapter 4 provides a framework for constructing system models which relate the tunable system parameters to the parameters that are of interest to the enterprise.

CHAPTER IV

*i*MANAGE: SCALABLE POLICY-DRIVEN SELF-MANAGEMENT

4.1 Introduction

Consider large systems that are integral parts of an enterprise's IT infrastructure. Examples of such systems include those supporting enterprise websites, or inventory management subsystems, or even the distributed information systems supporting a company's daily operations. Administrators managing these systems are not only expected to keep them running, but in addition, many such systems must meet certain processing constraints, be highly available, offer differentiated levels of Quality of Service (QoS), meet certain Service Level Agreements (SLAs), and may be subject to unforeseen demands. Unfortunately, even the occurrence of seemingly routine events like load changes, node and link failures, software patches, or modifications of certain environmental parameters can cause such systems to behave in unexpected ways, often resulting in their failure to meet current objectives. Given these facts and acknowledging enterprises' growing reliance on their computing infrastructures, solutions must be found for system self-management. These solutions must be driven by high level business goals, be open and receptive to administrators, cope with dynamic changes in requirements and conditions, and scale from small, tightly managed individual subsystems to large company-wide support infrastructures.

The existing tools and techniques for enabling self-management of enterprise-scale systems are insufficient because they are either too general or too specific. For instance, the state-of-the-art system management tools deployed at large enterprises include software suites like IBM's Tivoli, which is a systems management platform

and HP's OpenView (now combined with Mercury), which can be used for managing large-scale systems and networks[1]. These tools are equipped with methods for system monitoring and for graphically displaying system status to administrators. However, their functionality for automated symptom determination, reasoning about symptom causes, and taking appropriate corrective actions remains rudimentary, in part due to the lack of standards and more importantly, due to the general nature of these tools. In contrast, researchers have successfully embedded self-managing capabilities into specific well-defined subsystems like database backends [72], request schedulers for multi-tier web services [15] and others. To complement the self-management work being done for specific subsystems, several researchers have been focusing on issues like policy-specification language [21], model building techniques [11] and efficient monitoring schemes [4]. Similarly, there has been some excellent research in the domain of automating specific tasks that are required for enabling self-management. A particular effort of note is the work on automated problem diagnosis, presented in [19, 78]. The work focuses on using the monitoring data gathered from a system to detect service level objective violations and correlating the violation to earlier violations for gaining useful insights. While automating subsystems and problem diagnosis is important, these specific techniques must be combined into a comprehensive framework in order to be effective for complex systems.

The goal of our research is to develop abstractions and methods that help bridge the gap between (i) the excellent progress made in the general domain of self-management, like automation of well-defined subsystems or specialized techniques for self-management tasks vs. (ii) the more general challenges posed by managing more complex and/or larger IT infrastructures and applications. Toward this end, we build on such prior work for online system management, we adopt the use of online monitoring and behavior detection tools and techniques [20], and we endorse the use of ECA policies

[1]IBM, Tivoli, HP, OpenView and Mercury are registered trademarks of their respective owners.

to describe and build our self-management framework. To also address the broader management challenges posed by complex and dynamic IT applications and infrastructures, however, we propose a novel representation of the system state-space that is geared towards enterprise system self-management, and we develop new techniques for dealing with the problems of *scale*, *dynamism*, *tractability* and *trust*. Tractability here refers to an administrator's ability to understand current management actions undertaken by the system and to the system's ability to expose its reasoning for those actions. To achieve the goal of system manageability, our system, *i*Manage, offers the following tools:

- *A system modeling framework* - *i*Manage collects system parameters and metrics (collectively called *system variables*) into a single representation of the system state-space, and identifies which actions are available to change the system state.

- *A scheme for reducing the complexity of the system model* - Since a typical system model is too complex to be used or even properly constructed, our tools provide mechanisms to partition the state-space into smaller units that are easier to deal with. These *micro-models* allow us to more precisely model critical aspects of the system, and to more effectively develop policies.

- *Techniques for evolving system models* - Policies that are appropriate under one set of conditions may become invalid as operating conditions and the environment changes. *i*Manage provides techniques for evolving system models and policies, including methods to learn new policies and incorporate human knowledge and experience to refine the policies.

- *Techniques for quantifying our confidence in a system model* - In order for our system models to be useful, the system administrator must be able to

trust them. *i*Manage associates a confidence value with each proposed self-management action, and allows the administrator to both understand and use this confidence value when deciding whether to let the system manage itself.

In the following section we motivate the *i*Manage approach by describing certain subsystems and properties of the operational information system deployed by Delta Air Lines, one of our industry partners. Our interactions with the administrators and developers at that site have motivated much of this work.

4.1.1 Motivating Example

The Passenger Information Delivery System – PIDS (shown in Figure 22) – is a middleware developed at Delta Technology, Inc. to serve two important needs of the airline. First, it is responsible for managing the passenger data sourced from the airline's TPF mainframe. Second, it provides access to passenger information via events and service interfaces. The PIDS middleware, which according to estimates by Delta Technology processes around 9.5 billion events annually, ensures near real-time delivery of processed events to 'consumers' – programs that need to receive the events – and to a database of current booking and flight information used in activities like those in support of Delta's web site. PIDS collects data from all over the airline. While much of its information comes out of the airline's TPF-based Deltamatic Reservation and Operational Support System (OSS), additional inputs like gate information, information about weather, etc. arrive from airports throughout Delta's worldwide system. Further passenger information is provided by the reservation system. Finally, most planes generate and transmit their own landing time, which is provided to PIDS via FPES (the flight progress event system).

There are hundreds of variables associated with the PIDS system that capture the current state of the PIDS servers, the current load conditions, client specific metrics and several others. Some of these variables are enumerated in Table 6. A

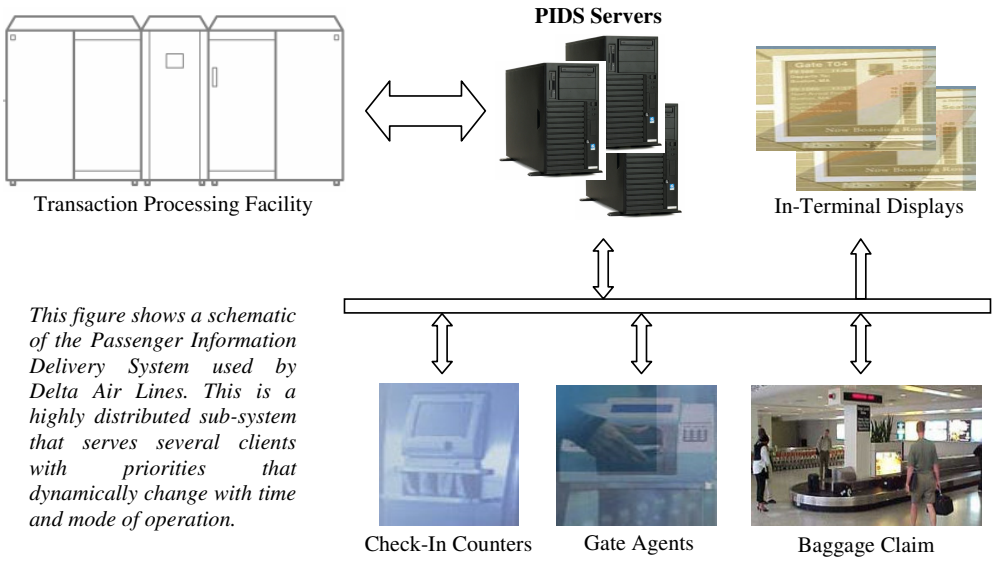

Figure 22: Some Interactions in the PIDS Middleware System

system administrator manages the system by virtue of having the ability to modify some of these variables, examples including the number of client service threads or the number of workflow service threads. More specifically, such modifications of state variables constitute the set of actions allowed for managing the system. The actions of a system administrator to respond to an event (like increased workflow processing delay) are based on his wisdom (mental model of the system behavior) and the prevailing conditions (values of different variables representing the current state). However, partial (and sometimes complete) failures of PIDS middleware are not uncommon, often resulting in delayed and/or canceled flights, and eventually leading to loss of revenue. Such failures can be attributed to the scale of the PIDS middleware and to the dynamic load conditions posed by the application domain.

The above example justifies our focus on the issue of scalability when designing our self-management framework. Moreover, in order to deal with the dynamic load conditions experienced by the PIDS middleware one must make use of self-management techniques that can continuously evolve. Finally, our interaction with the system administrators running the PIDS middleware motivated the need to keep the humans

Table 6: Some variables associated with the PIDS middleware

Variable	Description
	Global Variables
E2EL	The end-to-end latency introduced by the processing workflow.
ELPP	The average queuing delay at individual PIDS processing nodes.
CLIE	Number of cache access clients being served at any time.
ETTR	Expected time to recover from a failure.
EDRR	Events dropped in last 100,00,000 events.
CSTH	Client service threads at individual PIDS processing nodes.
WSTH	Workflow service threads at individual PIDS processing nodes.
NGAG	Number of active boarding gates
NBCA	Number of active baggage claim
NCIC	Number of active check-in counters
NOVR	Number of active overhead displays
	Gate Agent Variables
TTFD	Time to flight departure
DEST	Identifies whether a flight is domestic or international
NPAS	Number of passengers scheduled to board the flight

in the self-management loop and in control of the adaptation actions. This translates to the requirements for tractability and trust.

4.1.2 Road Map

The rest of this chapter is organized as follows. In Section 4.2 we present an overview of the overall approach, introduce the system state-space model used by the *i*Manage framework and describe the requirements for enabling self-management. Section 4.3 focuses on the specifics of our approach by describing the algorithms and techniques used by our framework, these include - the partitioning algorithm, the model building technique and the specifics of policy learning, adaptation and the confidence attribute. In Section 4.4 we present the evaluation of our techniques. Section 4.5 discusses the related work and finally, we conclude in Section 4.6 with some open problems for further research in this area.

4.2 Overview & State-Space Model

In this section we present an overview of our solution approach, which is followed by a formal description of *i*Manage's system state-space model and thereafter we present the requirements for enabling self-management of an enterprise-system. The formal model is used in the following sections to formally describe the various algorithms and techniques used by the *i*Manage framework.

4.2.1 Solution Overview

The *i*Manage framework for enabling self-management of enterprise-scale systems provides an abstraction of a system state-space, where each axis represents an identifiable system variable (e.g., end-to-end delay, throughput, etc.). The state-space model specifically identifies two sets of variables - one set contains the variables that determine the operational status of the system, and the other set contains the variables that can be modified to affect the state of the system. The first set is used for specifying the goals or SLAs and the second set is used to determine 'actions' that later become part of ECA policies for the system. In order to manage the system one needs to establish a model that connects the set of actionable or controllable variables to the set of goal variables. However, given the scale of the state-space for enterprise-scale systems and the fact that the system can exhibit different behaviors in different state sub-spaces, modeling the state-space is not straight forward. The *i*Manage framework utilizes a novel state-space partitioning scheme to deal with the problems of scale and heterogeneous system behavior. *i*Manage then makes use of tree augmented naive Bayesian networks or TANs to build 'micro-models' for each partitioned sub-space that results from the state-space partitioning algorithm. As a result, the system model becomes a collection of the 'micro-models' constructed for each sub-space. In case some goal violation is detected, the system model is consulted to arrive at new values for the set of action variables. In terms of policy the goal

violation becomes the *event*, the value of system variables at the time of violation become the *condition* and the assignment of new values to the set of action variables becomes the *action*. Since probabilistic models are used to arrive at a solution in case of goal violation, even the suggested policy actions are associated with a certain probability of bringing the system to a state of non-violation, and are enforced only when such probability or 'confidence attribute' exceeds the threshold set by a system administrator.

4.2.2 System State-Space Model

The following convention is used to describe the system state-space model. We use boldface capital letters such as, $\mathbf{V}, \mathbf{V}_\phi$ to denote sets, and assignment of values to variables in these sets are denoted by regular capital letters such as $\mathtt{V}_1, \mathtt{V}_2$. Similarly, we use boldface lower case letters such as, $\mathbf{v}_i, \mathbf{v}_j$ to represent variables that occur in the sets, and regular lower case letters such as, $\mathtt{v}_1, \mathtt{v}_2$ denote specific values taken by those variables.

We consider a system whose state can be represented by a set $\mathbf{V}$ of n variables $\{\mathbf{v}_1, ..., \mathbf{v}_n\}$, which are not necessarily independent. Out of these n variables the system's operational status (like failed, stable, unstable, etc.) can be determined by using only a subset $\mathbf{V}_\phi$ (an example of such variable would be the delay experienced by the users of an enterprise's website) of the state variables in $\mathbf{V}$. Therefore, $\mathbf{V}_\phi$ is the set of variables of interest as far as the system's operational status is concerned.

Furthermore, we associate the system with a set $\mathbf{A}$ of m action interfaces $\{\mathbf{a}_1, ..., \mathbf{a}_m\}$, such that an instance $\mathtt{a}_1$ of action interface variable $\mathbf{a}_i$ represents an action that can be invoked on the system. The invocation of an action $\mathtt{a}_1$ on a system state $\mathtt{V}_1$ is denoted by $\Omega(\mathtt{a}_1, \mathtt{V}_1)$, which possibly translates the system to a new state. The effect of invocation of action $\mathtt{a}_1$ on an instance of a system state-space variable $\mathtt{v}_1$ is similarly represented using $\omega(\mathtt{a}_1, \mathtt{v}_1)$. The above discussion is used to arrive at the following

definition of a deterministic action-variable pair.

> *A tuple* $(\mathbf{a}_i, \mathbf{v}_i)$ *is said to be a deterministic action-variable pair if* $\omega(\mathtt{a}_j, \mathtt{v}_k)$ *is known for all instances* $(\mathtt{a}_j, \mathtt{v}_k)$ *of* $\mathbf{a}_i$ *and* $\mathbf{v}_i$.

The set of all deterministic action-variable pairs of a system constitutes the set $\mathbf{D}$, and the set of all state-space variables that occur in any tuple in the set $\mathbf{D}$ constitute the set $\mathbf{V}_\alpha$, also called the set of controllable variables. The following lemma holds for all members of the set $\mathbf{V}_\alpha$.

> *If* $\mathtt{v}_1$ *and* $\mathtt{v}_2$ *are two possible values of the state-space variable* $\mathbf{v}_\alpha^i \in \mathbf{V}_\alpha$ *then there exists an instance* $\mathtt{a}$ *of* $\mathbf{a}_i$ *such that* $(\mathbf{a}_i, \mathbf{v}_\alpha^i) \in \mathbf{D}$ *and* $\omega(\mathtt{a}, \mathtt{v}_1) = \mathtt{v}_2$.

In order to manage a system, and affect its status, one needs to be able to deterministically modify the value of variables contained in $\mathbf{V}_\phi$. However, we only know of ways to deterministically modify the value of variables contained in $\mathbf{V}_\alpha$. Therefore, if one could discover a function χ that maps the space of variables of interest, $\mathbf{V}_\phi$ to the space of controllable variables, $\mathbf{V}_\alpha$ then one would be able to manage the system as described next. Let, $\mathtt{V}^{current}$ represent the current state of a system and $\mathtt{V}_\phi^{current}$ and $\mathtt{V}_\alpha^{current}$ represent values of the corresponding sets of variables $\mathbf{V}_\phi$ and $\mathbf{V}_\alpha$. Now, if the system needs to be translated to a new feasible state such that the variables of interest take the value $\mathtt{V}_\phi^{goal}$, then one should be able to determine $\mathtt{V}_\alpha^{goal}$ using the function χ and then use the set $\mathbf{D}$ to determine the actions required to change the value of variables in $\mathbf{V}_\phi$ from $\mathtt{V}_\alpha^{current}$ to $\mathtt{V}_\alpha^{goal}$.

Note that the set of variables in $\mathbf{V} - (\mathbf{V}_\phi \cup \mathbf{V}_\alpha)$ are not redundant and as we shall see in Section 4.3.1 they play an important role in determining the function χ. An example of such variable would be a measurement of number of disk-operations - such a variable is usually not a member of $\mathbf{V}_\phi$, which is used to determine acceptable system operational status; and this metric, in general, cannot be deterministically

affected by allowed system actions (e.g. allocating another disk-array). However, such variables may give hints about the actions to be taken to remedy a certain problem.

To put the above discussion in context, such a system model can be readily applied to the example discussed in Section 4.1.1. For example, the list of variables, enumerated in Table 6, constitute the set $\mathbf{V}$ of state variables for the PIDS system. The set of variables $\{E2EL, CLIE\}$ are the variables of interest as far as the operational status of the PIDS middleware is concerned and therefore constitute the set $\mathbf{V}_\phi$ (this corresponds to two of the several requirements imposed on the PIDS middleware -*the processing workflow should not introduce a delay of more than 1 second,* and *the system should be able to handle 3000 concurrent requests from the clients*). The set $\mathbf{V}_\alpha = \{CSTH, WSTH\}$ constitutes the set of variables that have action associations.

4.2.2.1 Limitations.

In the above discussion we assumed that all of the variables that constitute the system state-space are known. This is not true for systems where due to considerations like monitoring overhead and complexity some of these variables might not be monitored. However, the probabilistic modeling techniques used by our framework are able to perform sufficiently well even when some of the variables are not listed as members of the system state-space, or are not monitored by the system. One must note that failure to include some important state-space constituents may lead to a system model which might not be manageable.

The second limitation arises from the fact that the function χ might return multiple possible instances of the set $\mathbf{V}_\alpha$ corresponding to the goal state represented by $\mathtt{V}_\phi^{goal}$. For example, if $\mathtt{V}_\phi^{goal}$ corresponds to reduction in end-to-end delay for a three tier web-server, then there may exist multiple actions like increasing the number of front-end servers or upgrading the backend database server that may lead to reduction

in end-to-end delay. Our probabilistic techniques will suggest the solution which has the highest probability of resolving the problem without any guarantees about the efficiency or optimality of the solution. This opens up the possibility of a difference between 'manageable' and 'efficiently' or 'optimally' manageable system. However, in this chapter we will limit ourselves to the concept of manageability.

4.2.3 Enabling Policies

There are certain requirements that should be met by any system to become eligible for policy-driven self-management. Firstly, the system should be able to measure and export the current value of variables that constitute the state-space for the system. One can think of this as 'dials' on a control dashboard used for managing a very large system. Secondly, the ability to modify some of the variables is also central to the idea of policy enablement. One can similarly think of this capability as the 'knobs', which can deterministically change the value displayed on some 'dials'. In terms of our system state-space model, the variables represented by 'dials' are the variables in the set $\mathbf{V}$. The variables which have an associated 'knob' constitute the set $\mathbf{V}_\alpha$. The 'knobs' can in turn be used to take actions specified using $\omega(\mathtt{a}_i, v_\alpha^j)$.

In order the enable policies in a policy-ready system, we need to have a way for representing the policies, mechanisms that discover and learn policies at runtime, ways to enforce policies and techniques for keeping the policies updated for the current system environment. The following sub-sections briefly describe our approach to handling these issues. Some of these issues will later be dealt in detail in Section 4.3

Policy Specification - We use a modified form of the well accepted event-condition-action (ECA) format for specifying the policies. The ECA specification is very useful when it comes to enforcing policies for any system. However, we extend the specification to include a confidence-attribute that is related to the probability of the policy having a desired effect when the action specified as part of the policy is taken under

appropriate conditions. The *event* in our policy description is a change in the value of some variable(s) in $\mathbf{V}_\phi$. The *condition* that triggers the action associated with the policy is specified over the set of variables in $\mathbf{V}$. The *action* is similarly specified as the modification in the value of some variables contained in $\mathbf{V}_\alpha$.

Policy Discovery - We believe that all policies cannot be specified and that the system may need to discover some policies on the fly. We use a novel state-space partitioning scheme, described in Section 4.3.1 to first reduce the system state-space under consideration at any instant. Then for each partition we make use of greedy algorithm to discover the most important variables from the set $\mathbf{V}_\alpha$ (i.e. the right knobs). We finally make use of Bayesian networks to build 'micro-models' of the the state-space corresponding to each partition, thereby enabling us to find the values to which the 'knobs' should be adjusted to. We elaborate on these techniques in the following sections.

Policy Enforcement - The interfaces that export the current value of system variables are continuously monitored for any changes. Changes in the values of some variables may cause some policy to evaluate its condition, and if the condition evaluates to true, the action specified as part of the policy is taken. In simple words, when a problem occurs (i.e., the value on some dial signals something bad) the self-management subsystem tries to (1) find the 'right-knobs' and then (2) adjusts them to some appropriate new values. The enforcement of any policy is also contingent on the confidence-attribute, which should be more than a system-wide threshold set by the system administrator. This gives the administrator a control over the degree of self-management.

Policy Refinement - Policies that are either specified or are learnt by the system may need to be changed because the conditions under which such policies are valid may change with time. An instance of this would be the addition of more nodes to the network underlying the operational information system. Such instances may lead to

changes in threshold values that trigger an action specified as part of the policy. Our techniques are able to keep track of such changes in the environment and in response, they suitably modify the policies.

4.3 Solution Approach

The system state-space model proposed in Section 4.2.2 showed that $\mathbf{V}$, $\mathbf{V}_\phi$, $\mathbf{V}_\alpha$, $\mathbf{A}$, $\mathbf{D}$ and χ are the parameters that should be known for arriving at a self-management solution for a system. One can safely assume that for most of the systems the sets $\mathbf{V}$, $\mathbf{V}_\phi$, $\mathbf{V}_\alpha$, $\mathbf{A}$ and $\mathbf{D}$ are known apriori. This implies that the system variables, the variables of interest and the deterministically modifiable variables along with the ways to modify them are known. This is true for enterprise-scale systems where the system variables like number of network nodes, link capacities, etc. are known. Similarly, the variables of interest like end-to-end delay are also known apriori. Lastly, one knows of variables like allocated buffer-length at network nodes which can be deterministically modified by changing some system parameters. The problem is to find the function χ, and this means that we need to find a way to model the system. Remember that the function χ relates the variables in $\mathbf{V}_\phi$ to the variables in $\mathbf{V}_\alpha$ and the function χ can change for different values of variables in $\mathbf{V} - (\mathbf{V}_\phi \cup \mathbf{V}_\alpha)$. Once the function χ has been determined for the system state-space, one can easily find and/or adapt the actions that form part of the policy specification.

However, building a model (i.e., determining χ) for understanding the behavior of an enterprise-scale system is difficult. This can be attributed to the fact that in such systems there are a large number of variables (e.g., bandwidth, workload, queue length at servers, etc.), each one of which can potentially affect the state of the system and more often than not these variables also interact amongst themselves. For example, in a certain sub-space of the system's state-space the bandwidth between participating nodes may be the bottleneck and any modification to the priority of

processes may have little or no effect on the observed performance. The situation may similarly be reverse for some other system state sub-space where server capacity may be the bottleneck and any modification to the inter-node bandwidths may have no effect on the observed performance. The two insights that follow from the above discussion are that:

1. Finding a single function to model the entire state-space of an enterprise-scale system might lead to very crude and incorrect system models.

2. There exist system state sub-spaces where the effects of certain variables can essentially be ignored from the system model.

The above discussion motivates the need to partition the system state-space. The following sub-section elaborates on the specific requirements for the partitioning scheme and then describes the partitioning algorithm in detail.

4.3.1 System State-Space Partitioning

The aim of our partitioning scheme is to create system state-space partitions such that:

- The involved system variables exhibit some homogeneity in their behavior inside the partition, which is beneficial for building the system model.

- The number of 'knobs' required to manage the system within the partition is minimized, which is beneficial for the purpose of learning and adapting the actions specified as part of policy.

To incorporate the concept of partition homogeneity we create partitions such that operational states contained in the partition are close to each other. Note that partition homogeneity corresponds to macro-level states of the system, for instance in one partition the underlying network may be the bottleneck (making server capacity

redundant) while in some other partition the server capacity may be the bottleneck (similarly making the network capacity redundant). In order to minimize the 'knobs', we want to ensure the partitions are created such that the 'knobs' needed in one partition are possibly not needed in the other. This corresponds to making partitions which are orthogonal to each other. The partitioning algorithm employed by our framework is described next.

4.3.1.1 The Partitioning Algorithm

A system state can be defined as the binding of appropriate values to the variables contained in the set $\mathbf{V}$. The partitioning algorithm aims to partition many such observed system states to achieve the objective mentioned in the previous section. We define a partition to be a collection of observed system states. A partition inherits the sets $\mathbf{V}$ and $\mathbf{V}_\phi$ from the system state-space but the sets $\mathbf{V}_\alpha$, $\mathbf{A}$ and $\mathbf{D}$ can vary between the partitions.

Let S be the observed operational states contained in the initial system state-space partition for which $\mathbf{D}$ defines the association of action interfaces in $\mathbf{A}$ with the variables in $\mathbf{V}_\alpha$. For simplicity the discussion here assumes that it is possible to define a measure of normalized distance between any two operational states. Techniques for doing such operations exist and interested readers may refer to well-known techniques like Mahalanobis distance [55]. We define an operator $\delta_{\mathbf{R}}$ over a pair of operational states from a partition, which finds the normalized distance between the two operational states considering only the dimensions contained in the set $\mathbf{R}$, where $\mathbf{R} \subseteq \mathbf{V}$. We also define the operation θ over a pair of operational states from a partition. The operation θ finds the number of places in which the two states differ, considering only the dimensions corresponding to the set $\mathbf{V}_\alpha$ for the partition under investigation. Finally, we define

$$\upsilon(s_1, s_2) = \eta \times \delta_{\mathbf{V}}(s_1, s_2) + \mu \times \theta(s_1, s_2) \tag{5}$$

where, η and μ can take values from the range [0,1] and these are used to configure υ for weighted distance and orthogonality. To evaluate if we need to partition a given system state-space P, we try find a subset $\mathbf{V}'_\alpha$ of $\mathbf{V}_\alpha$ such that

$$\sum_{\forall s_i, s_j \in S} \delta_{\mathbf{V}_\alpha - \mathbf{V}'_\alpha}(s_i, s_j) \leq \Delta_{max} \tag{6}$$

$$|\mathbf{V}'_\alpha| \leq f \tag{7}$$

where Δ_{max} is a user defined parameter that represents the maximum allowed representation error for the controllable variables and f represents the maximum number of actions that can be used to manage the system in the given partition. We employ a greedy approach for finding $\mathbf{V}'_\alpha$, i.e., we add the member of $\mathbf{V}_\alpha$ to $\mathbf{V}'_\alpha$ which causes the greatest reduction in the L.H.S. of the equation 6. We repeat the above process until the L.H.S. becomes lesser than Δ_{max}, at this point we look at the cardinality of the set $\mathbf{V}'_\alpha$ - if the cardinality is less than f we do not partition the system state-space, otherwise we proceed to partition the system state-space. The $\mathbf{V}'_\alpha$ so determined becomes the $\mathbf{V}_\alpha$ for the partition. We start by finding a pair of states s_1 and s_2 from the set of all such pairs contained in the set S such that $\upsilon(s_1, s_2)$ is maximized. The pair s_1 and s_2 acts as the seed for the two new system state sub-spaces S_1 and S_2 that will be created. We then iterate through the remaining operational states in the set S, adding the operational state s_i to S_1 if $\delta_{\mathbf{V}}(s_i, s_1) \leq \delta_{\mathbf{V}}(s_i, s_2)$. One can alternatively use the centroid of existing operational states in the evolving partitions to determine the membership. Once the two new partitions S_1 and S_2 have been created, we find the set $\mathbf{V}_\alpha$ for them using the greedy approach described above. If the criteria defined by Δ_{max} and f is not met by any partition then we repeat the above scheme for that partition. We now enumerate the advantages of the partitioning scheme for the purpose of enabling enterprise system self-management.

- *Assists in Problem Diagnosis.* The system might migrate through a series of

system state 'partitions' before ending in an unacceptable state (e.g. SLA violation). The path followed by a system before a failure may contain information about the events that may have led to a failure, and can therefore assist in problem diagnosis and constructing complex policies.

- *Simplifies Problem Resolution.* If a system enters an unacceptable state during its operation then the model corresponding to the partition to which this unacceptable state belongs can possibly be used to arrive at a resolution to the problem.

- *Reducing Monitoring Overhead.* The partitions that are created by our algorithm allow us to ignore a subset of variables when the system is operating in that partition. This can potentially allow us to monitor such variables at reduced frequencies. However, we have not fully explored this possibility.

- *Simplifies Policy Learning.* Our approach intelligently reduces the space of possible actions that could be taken in response to an event. This greatly simplifies the process of correlating the events to actions for the purpose of determining ECA policies.

Once the system state-space has been partitioned, we build a system *micro-model* corresponding to each partitioned sub-space. A system model in our framework consists of several micro-models each one of which models a sub-space of possible system states. The micro-model to be applied is determined based on the current state of the system. Since, we attempt to model only a small partition of the entire system state-space at a time we are able to build models even for systems with a very high number of variables. This makes our approach highly scalable. A similar approach was presented in [78], which made use of an ensemble of probabilistic models to detect SLO violations, and was shown to perform significantly better than the approach which used a single monolithic model to detect violations. The approach works by

adding new models when the existing models do not accurately capture the current system behavior.

4.3.2 Building System Micro-Models

We want to create *micro-models* such that they can predict the values for the variables in $\mathbf{V}_\alpha$ given the values for the variables in $\mathbf{V} - \mathbf{V}_\alpha$. Here we take advantage of the fact that our system state sub-spaces have a reduced dimension in terms of controllable variables. For all the variables in $\mathbf{V}_\alpha$ we exhaustively enumerate all the possible values and create a new variable $\mathbf{c}$ which can take values corresponding to such an exhaustive enumeration. For example, if $|\mathbf{V}_\alpha| = 2$ and each variable in $\mathbf{V}_\alpha$ can take 3 discrete values then $\mathbf{c}$ can take 9 distinct enumerated values. We assume that the controllable variables take discrete values and if the variable is continuous, one can use existing techniques to discretize continuous data (actually the Bayesian modeling techniques, which are referred to in this chapter make use of such techniques). The system state space $\mathbf{V}$ can now be represented as $\{\mathbf{c}\} \cup (\mathbf{V} - \mathbf{V}_\alpha)$.

To find the best value from the variable $\mathbf{c}$ which helps translate the system to a desired state we resort to making using of probabilistic modeling techniques. We use a variant of Bayesian network [35] called the Tree Augmented Naive Bayes [28] or TANs to probabilistically model the system state-space. A Bayesian network is represented as an acyclic graph whose vertices encode random variables and the edges represent statistical dependence relations among the variables and local probability distributions for each variable given values of its parents. The main advantage of using a Bayesian network (or one of its variants) is that their representation provides and easy way to inspect the relationships between the involved variables. This allows an expert to embed his knowledge or the common wisdom into the self-management framework by proposing an initial model, which can be further refined using learning techniques. Furthermore, by simple inspection an expert can single out any faults in

the learnt system model. Our choice for making use of TANs was driven by the fact that unrestricted forms of Bayesian network are computationally very costly to build as they need to evaluate all the dependencies amongst the set of random variables. A TAN, on the other hand allows only a tree structured dependence amongst the set of random variables (other than the class variable) and is therefore cheaper to build and has been shown to perform almost as well as the unrestricted version. A TAN model when used as a classifier is able to determine the following probability

$$p = Pr(\mathbf{x}|\mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_n) \tag{8}$$

for the set $\{\mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_n, \mathbf{x}\}$, from a given training set. The variable $\mathbf{x}$ assumes a special status in this equation and is called the class variable and the other variables are called the attributes.

To create the micro-model we designate the newly formed variable $\mathbf{c}$ as the class variable and the remaining variables, i.e., the set of variables in $\mathbf{V} - \mathbf{V}_\alpha$ are designated as attributes. The resulting micro-model is able to determine the following probability.

$$p = Pr(\mathbf{c}|\mathbf{V} - \mathbf{V}_\alpha) \tag{9}$$

The above equation determines the probability of the variable $\mathbf{c}$ taking a certain value given the values of the variables in the set $\mathbf{V} - \mathbf{V}_\alpha$. This procedure for achieving a desirable and feasible system state is as follows. Let, $\mathtt{v}^{current}$ represent the current system state and $\mathtt{v}_\phi^{goal}$ represent the new desired values for the set $\mathbf{V}_\phi$. To find the values of variables in $\mathbf{V}_\alpha$ that can possibly lead to the goal state, we create the set $\mathbf{V}' = \mathbf{V} - \mathbf{V}_\alpha$. We create an instance $\mathtt{v}^{goal}$ of the set $\mathbf{V}'$ by assigning the corresponding values from the set $\mathtt{v}^{current}$ and thereafter resetting the values corresponding to the set $\mathbf{V}_\phi$ using the values from $\mathtt{v}_\phi^{goal}$. We then use the instance $\mathtt{v}^{goal}$ to find the instance $\mathtt{c}^{goal}$ of variable $\mathbf{c}$ that maximizes Equation 9. The values of $\mathbf{V}_\alpha$ corresponding to $\mathtt{c}^{goal}$ so determined are used as the new values for controllable variables.

4.3.2.1 Discussion.

Note that the state depicted by $\mathtt{V}^{goal}$ may not exist in a real system. This is because the variables that are contained in $\mathbf{V} - \mathbf{V}_\phi \cup \mathbf{V}_\alpha$ inherit their values from the state instance $\mathtt{V}^{current}$, and it may so happen that when the system translates to the new goal state, the values for variables other than the variables of interest and controllable variables may also change. However, experimental results presented in Section 4.4 show that the predicted values of $\mathbf{V}_\alpha$ are mostly able to achieve the goal state. This can be attributed to the fact that attribute discretization adds some degree of tolerance causing some smaller changes not to be reflected until they occur at the points where discretization partitions the continuous data space. Another important consideration for future work may be the consideration of the magnitude of change in the values of the variables in $\mathbf{V}_\alpha$. A solution that requires smaller change in magnitude may sometimes be preferred over the most probable solution.

4.3.3 Policy Learning, Adaptation & Confidence Attribute

The system state space model and the micro-model play a central role in supporting the task of policy learning. The high-level directives or goal statements are described over the set of variables contained in the set $\mathbf{V}_\phi$. For example, a high-level directive like $\mathtt{delay} < 20msec$ can be used to learn the corresponding policy using the procedure described next. The framework instantiates a trigger for capturing $\mathtt{delay} \geq 20msec$ which acts as the *event* in terms of policy. If at any time the *event* occurs the current system state $\mathtt{V}^{current}$ is used in conjunction with system sub-space micro-model to arrive at a corrective *action*. The event, the current system state (condition) and the corrective action are recorded as a policy.

In a dynamic system the micro-models may evolve with time. This may cause some learned policies to become invalid with time because the corrective actions that were determined using an earlier version of the model may not be applicable any

more. Policy adaptation requires periodic evaluation of the actions specified as part of the learned policies.

The confidence-attribute associated with each policy helps us to deal with the issue of administrator's *trust* in our self-management framework. The confidence-attribute for a suggested self-management action is equal to the probability p determined using the Equation 9. The system administrator can declare a threshold value to have control over the policies that will be enforced. Only the policies with a confidence-attribute greater than the threshold value are autonomically enforced by the self-management framework.

4.4 Experiments

Our goal was to study the suitability of our techniques in managing large enterprise-scale systems where a large number of variables can potentially affect the state of the system. In this section, we present our findings based on simulation experiments under a variety of workloads and operating conditions. Our techniques, for instance, were able to detect bottleneck nodes in our simulation of a PIDS-like middleware and were able to avoid several SLA violations that would have otherwise occurred. We start with a description and validation of the simulator testbed, which is followed by a brief description of the workload and evaluation metrics. We present our experimental results starting from Section 4.4.3.

4.4.1 Simulator Testbed

We wanted to evaluate our techniques for self-management using applications that are representative of the ones used by large enterprises. We evaluated the possibility of using well-known benchmarks and real-enterprise applications for putting our techniques to test. However, we soon realized that the applications available to us in our lab environment (like RUBiS [60] and an implementation of industrial middleware from Delta Technology [23]) were not instrumented well enough to sense and

actuate a sufficiently large set of variables, and often changing any environment parameter (like maximum number of worker threads, number of MySQL connections, memory allocations and MySQL cache size) required restarting the application for the changes to take effect. Of course, in order to use our techniques, these systems could be enhanced to provide more monitoring and dynamically tunable parameters. Furthermore, it was not possible for us to make use of such applications for a large-scale (say 500 underlying nodes) evaluation of our techniques.

In order to overcome the problems mentioned above we decided to design a well instrumented simulator for simulating a large system implementing service oriented architecture (SOA). An implementation of SOA contains a set of services running on a distributed network of nodes that can be invoked by sending a message to the service, messages may or may not be generated as a result and if the messages are generated they may be forwarded to the source, to a sink or to some other service(s). The PIDS middleware described in Section 4.1.1 can be implemented as a SOA. Our SOA simulator consists of four main components - server, service, network-link and client. A server represents a processing facility with a limited number of cycles per second, a limited memory, connections to other servers and ability to throttle server frequency at the expense of more power. A service represents a software which accepts certain types of messages, possibly generates some messages in response and determines the server cycles that will be used to process a certain message type given the available memory. There may be more than one service running on a server and they may have different priorities. A network-link has an associated bandwidth, delay, and cost per unit of data transmitted. Finally, a client represents a source or a sink for the messages. An event source has a rate of generating events that can vary with time. A sink measures the incoming event-rate, the average delay for update propagation and the current delay measured over a recent window.

A simulation can be started by providing a network topology which instantiates

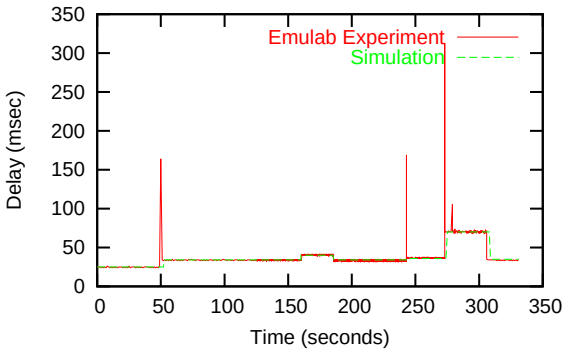

Figure 23: Validation of the SOA simulator against an emulation at Emulab

the basic distributed network of servers and network-links, this is followed by addition of some services for processing messages and some clients. The simulation is run for a pre-specified amount of time and it dumps the state at configurable regular intervals. We make use of these state dumps to evaluate to build system models and then use these system models to arrive at policies for managing the simulated system. The network topologies for the simulations were generated using the GT-ITM [76] generator.

4.4.2 Simulator Validation

To validate our simulator we compared the measurement attained by simulation with the ones attained using the same experimental setup on our Emulab [26] testbed. The experimental setup consisted of a 13 node topology and an event processing graph comprised of 3 sources, 2 services and 2 sinks. On the Emulab testbed we created the specified 13 node topology and then made use of the *i*FLOW middleware [45] to set up the event processing graph. We instantiated the same setup using our simulator. The services were configured to take a specified amount of time for processing the incoming events depending upon the incoming event type, and server load. We measured the event propagation delay between a source and a sink under a variety of variations which included events that take different processing times, variation in event rate

from sources and change in event-size. The same event workload was used for both the Emulab testbed and the simulator. The measurement of event propagation delay for both the Emulab testbed and the simulator is shown in Figure 23. Our simulator was able to closely follow the behavior of the real emulation testbed for the same experimental setup, thus paving the way for simulations at a larger scale.

4.4.3 Microbenchmarks

We ran two simulations with 8 and 32 servers, each for 4 simulated hours. The two simulations dumped 71 and 227 state variables, respectively every 30 seconds. During the course of simulation we kept modifying the system conditions like the event rates from the sources, modifying the server frequencies, using alternate high or low-cost links to the destination and changing the priorities for various services running at a server. We also ran the simulation for another 20 minutes, dumping data at every 30 seconds to evaluate the accuracy of generated models. We collected 3 such sets of observations.

The first experiment focused on determining the effect of partitioning parameters Δ_{max} and f on the number of partitions that are created for a given system state-space and the average number of controllable variables that appear across the partitions. The results obtained by using one set of observation from the simulation described above are shown in Table 7. The table enumerates results from two set of simulations described above which generated 480 observed system states each. The results show that our techniques were able to significantly reduce the average number

Table 7: Effect of partitioning parameters on $|\mathbf{V}_\alpha|$ & number of partitions

		Original		**Partition**	
f	Δ_{max}	$\|\mathbf{V}\|$	$\|\mathbf{V}_\alpha\|$	avg $\|\mathbf{V}_\alpha\|$	partition count
3	0.1	71	11	2.8	5
4	0.2			4.0	3
3	0.1	227	31	2.7	7
4	0.2			3.8	5

Table 8: Comparison between the accuracies (in %age) of single and micro-models

Data Set	Model Type	
$\|\mathbf{V}\|$	Single Model	Micro-Model
71	89.4 ± 2.8	92.3 ± 3.2
227	86.3 ± 1.9	90.7 ± 2.3

of controllable variables. For example our partitioning scheme was able to able to achieve a 90% reduction in number of controllable variables per partition for a system state-space with 31 variables. As far as the number of partitions is concerned, it is an important contributor towards the scalability of our techniques. However, a very high number of partitions may lead to partitions that may have a very sparse population leading to bad system models. The number of controllable variables are required to be low for our techniques to be effective as long as the manageability of the system partition is maintained, which can be controlled by setting a low value for Δ_{max}.

The next experiment was conducted to examine the effect of partitioning on the accuracy of the system models. To construct a single system state-space model for the set of observations collected earlier we proceeded as follows. We eliminated any $\mathbf{V}_\alpha$ from the set if it did not change its value during the simulation run. This reduced $|\mathbf{V}_\alpha|$ to ≈ 5 and ≈ 11 for the simulations with 71 and 227 variables, respectively. Notice that even with a discretization factor of 2 for each variable, the simulation with 227 variable had $2^{11} = 2048$ possible values for the variable $\mathbf{c}$. In the real world this translates to the confusion of which 'knobs' to turn to fix the system. Using the technique described in 4.3.2 we then constructed the single system models. For building micro-models corresponding to the partitioned sub-spaces we did not have to perform the pruning of the set $\mathbf{V}_\alpha$ as the partitioning algorithm takes care of removing the redundant members from the set $\mathbf{V}_\alpha$. The micro-models were then constructed for each of the partitioned sub-space. We used the generated models to predict the value of controllable variables given the value of other system variables from the test data set. Results reported in Table 8 show that the specialized micro-models work

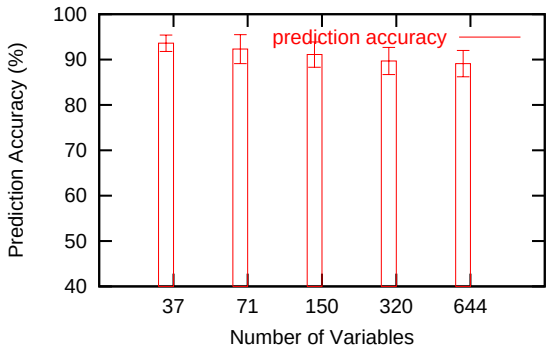

Figure 24: *i*Manage's scalability with number of observed system variables

better than a single model at correctly predicting the values of variables in $\mathbf{V}_\alpha$.

To examine the effect of the number of observed system variables (i.e. $|\mathbf{V}|$) on the accuracy of predicting the right values for the variables in $\mathbf{V}_\alpha$ we conducted the following experiment. We used our SOA simulator to simulate systems that had 37, 71, 150, 320 and 644 variables that could be observed. Each system was simulated under varying workload conditions with appropriate corrective actions being taken at several points in the simulation. Simulation time was proportional to the number of variables being observed for that system. The smallest system with 37 variables was simulated for 1 hour simulation time. We used our techniques to build models for each of the systems and then used 10 minutes of generated test data to calculate the prediction accuracy corresponding to each model. We repeated the experiment 3 times. Results shown in Figure 24 show a slight decrease in prediction accuracy with the increase in number of observed system variables. However, the prediction accuracy only shows a linear trend in decrease as the number of variables are increase exponentially. We acknowledge that the results obtained may be highly dependent on the training set and the test data set that were generated during the simulation.

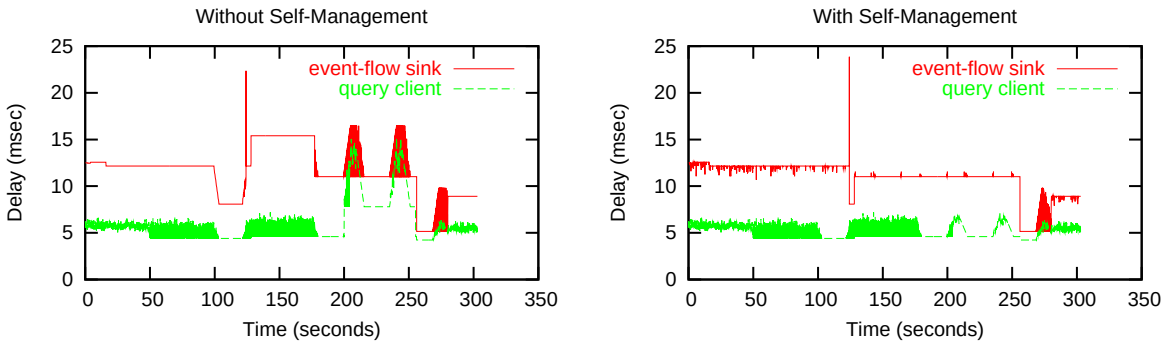

Figure 25: Delays from the SOA simulator with and without self-management

4.4.4 Evaluation of the Self-Management Framework

The next set of experiments was conducted to evaluate the end-to-end efficiency of our framework in managing large-scale systems. We study the impact of suggested self-management actions and the confidence attribute on the end system metrics at runtime. The simulations were conducted for a system with 227 variables and consisted of 32 simulated server nodes.

The simulation setup consisted of an event-flow that contained 3 sources, 2 services and 1 sink, and 2 query response services which received a stream of queries from a co-located client. Each event-flow service was located on a separate server but shared the server with another query response service. The variables that could be modified included the priority of the event-flow service thread, the priority of the query service thread and the frequency of the server. Dynamic workload conditions were simulated by varying the event rates from the sources and the query clients. The metrics of interest included $\mathbf{delay}_{flow}$ and $\mathbf{delay}_{query}$. The goals for the simulation run were specified as $\mathbf{delay}_{flow} < 12.5msec$ and $\mathbf{delay}_{query} < 7.5msec$, and both the threads were assigned the same priority. Figure 25 shows the delay observed at the event-flow sink and at one of the query client with and without self-management. Our techniques were mostly able to avoid any violations of the specified goals. The confidence threshold for this experiment was set to 85.0%.

We next conducted an experiment using the above setup to examine any unwanted behavior that may happen due to low confidence-threshold. When the confidence threshold was reduced to 75.0%, we actually observed delays at the event-flow sink that were more than the delays observed without the self-management in place. Confidence thresholds even lower than 75.0% made the system behave erratically when self-management was turned on. This was corroborated by re-examination of some of our earlier data used for prediction accuracy experiments. 90.0% of the predictions that led to false predictions had a confidence-attribute lesser than 65.0%. These findings can be attributed to the use of probabilistic models by our framework. A low-confidence threshold means that there is possibility that a certain other assignment of values to variables in $\mathbf{V}_\alpha$ also has a high probability of occurrence. This may lead to two assignments having almost the same probability of occurrence leading to a higher chance of erroneous choice of assignment by the system.

4.5 Related Work

4.5.1 Policy Research

There has been much work in the domain of using policies for simplifying the management tasks associated with system administration. Over the last decade, researchers, both in academia and industry, have focused on issues like policy specification languages [21], frameworks [33, 71] and toolkits [52]. The research presented in this chapter builds on the work done in the above mentioned areas and is a logical next-step, as the focus is on applying the policy-research to the management intensive domain of enterprise-scale systems. The policy research in the domain of automated network management that deals with issues like security, access control and other associated management tasks [75, 58] justifies our stand on studying the impact and application of policy research to another rich domain. More recently, researchers have

started evaluating the pros and cons of applying policy research for managing IT systems at large business enterprises [14]. This research, which is in its nascent stages, promises to provide systems that will manage themselves in accordance to high-level business goals [32]. The issues concerning human expertise and policy representation have also been explored in a recent paper [39].

4.5.2 Autonomic Computing & Self-Managing Systems

The task of implementing self-managing systems is a multi-step process in which policies can play an important role. Policies are a way to dictate the behavior of a self-managing system. This is in line with the vision of autonomic computing - 'to design computing system that can manage themselves given high-level objectives from administrators' - as described in [40]. There has been much work in the domain of enabling self-management for a wide variety of systems. The SLA-based approach to manage systems has been explored by a number of researchers [77]. In our prior work, we had focused on enabling self-management capabilities for distributed data stream systems [49, 45]. Some researchers have also explored the use of rule-based self-management approach for managing applications [11]. The use of utility-functions for self-management has also been explored in specific reference to event-based systems [12], and an interesting take on aggregate utility-functions is presented in [8]. It turns out that defining utility-functions for enterprise-scale applications is a tough task because it may not be possible to mathematically model all the factors that can potentially affect the state of the enterprise system.

4.5.3 Bayesian Networks & Problem Diagnosis

Bayesian networks or the Belief networks have found applicability in a number of AI domains, and they represent one of the best classification tools available to researchers. A tutorial on Bayesian networks is presented in [35]. Several specializations of Bayesian networks have been proposed in literature, the most important ones

being the Naive Bayes [24] and the Tree Augmented Naive Bayes or the TANs [28]. In reference to applying Bayesian networks for modeling computer systems, a very innovative approach for correlating instrumentation data to system states is presented in [19]. This was later extended in [20] to develop signatures that could be used to more efficiently correlate the SLA violations that may occur in a system. More recent work in this domain makes use of an ensemble [78] of system models for the purpose of problem diagnosis. The work presented in this chapter not only detects possible violations of higher level goals by the system but also suggests appropriate corrective actions to arrive at a solution for the problem.

4.6 Summary

In this chapter we described a system modeling framework that collects the system parameters and metrics into a unified abstraction, which we call the system state-space, and identifies the actions that can be used to manage the system. To deal with complex system state-spaces, typical of enterprise-scale systems, we presented techniques that can be used to reduce the complexity and to more precisely model critical aspects of the system, and to more effectively develop policies. Additionally, *i*Manage has capabilities for dealing with dynamic environments and for letting the administrator incorporate human knowledge and experience to refine policies. Finally, the confidence-attribute associated with the actions, which constitute the policies learnt by *i*Manage framework, allows the administrator to fine-tune the enforcement of such policies.

In real enterprise settings, a single end-to-end system is composed of multiple components. Any technique that decomposes the system level SLA or goals to per-component SLAs or goals would not only simplify the management of such infrastructures, but it would also add a new dimension to the scalability of self-management techniques. Once such per-component SLAs have been determined one can use

*i*Manage-like techniques or other time tested techniques (like the ones that exist for database tuning) to enable self-management for that component. The following chapter describes in detail our research on SLA decomposition.

CHAPTER V

SLA DECOMPOSITION

5.1 Introduction

The need for increased automation, better integration with internal processes and flexibility in interacting with external partners is creating increasingly complex IT systems and applications in today's large enterprises. Examples of such systems include those supporting enterprise websites, inventory or revenue management subsystems [3], and distributed information systems supporting a company's daily operations [31]. Typically, such systems are constructed as collections of components and/or independent software artifacts like web-servers, database systems, and local or remote application services, which interact with each other in many unpredictable patterns when providing the functionality required by the enterprise end users. Further issues include dynamic changes in resource usage and availability, caused by natural system behaviors and failures. Other factors such as resource migration and consolidation also contribute to such behaviors in today's virtualized enterprise. As a result, even when the behavior of constituent components can be well characterized and controlled, it is typically intractable to precisely characterize or limit the dynamic behaviors of the composed enterprise systems and applications. This intractability makes it difficult, if not impossible, for system administrators to efficiently achieve conformance to Service Level Agreements(SLAs) with internal or external enterprise partners. In many cases, systems are over-provisioned to meet SLAs. One such example is the extensive use of over-capacity to guarantee time limits on search requests for the flight search services offered by one of our industry partners, Worldspan [56].

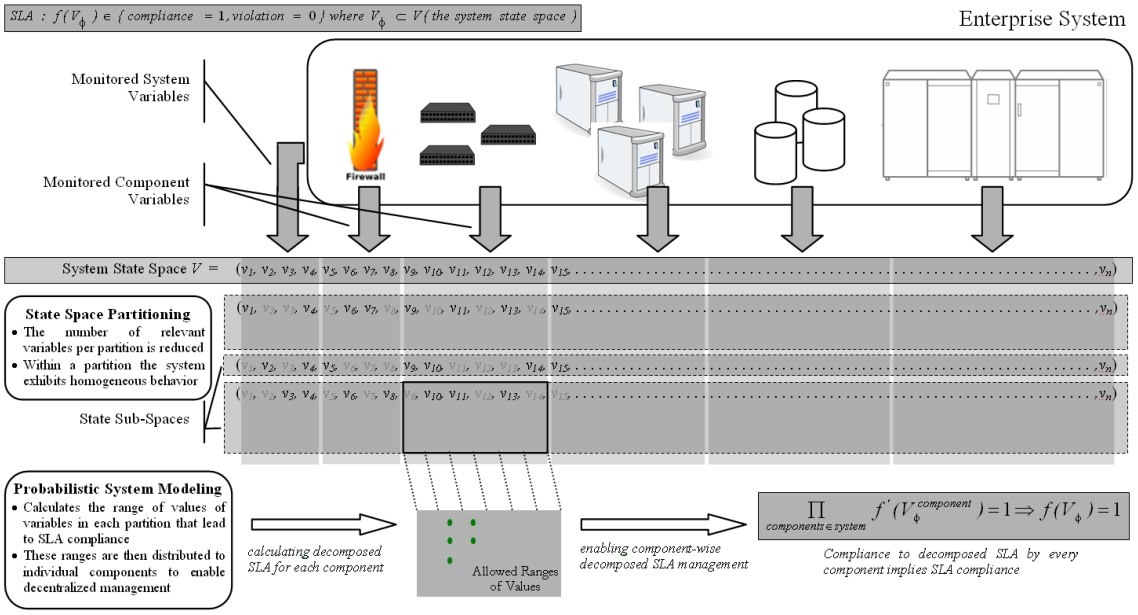

Figure 26: Determining Component-Level Objectives from Service-Level Agreements

This chapter addresses the complexities arising from dynamic component interactions in large-scale enterprise applications by deriving component-level objectives from high level business goals such as SLAs. The solution is based on the assumption that a system's constituent components or subsystems can be individually monitored and managed to the extent needed to attain desired runtime component behaviors. Our approach, as shown in Figure 26 starts by constructing meaningful state-spaces for enterprise system, based on runtime monitored variables. Scalability and manageability are achieved by dynamically partitioning the enormous state-space generated from typical application runs into smaller homogeneous sub-spaces. These homogeneous sub-spaces, which are representative of typical application behavior, are then modeled using probabilistic modeling techniques to create micro-models. For example there may be a micro-model capturing the steady state-behavior of the system, while another micro-model may correspond to the system state in which the back end is being updated by a new batch of updates. These micro-models are then used to derive component-level objectives that characterize a component's contribution to the high

level goal. In this way, we dynamically determine the possible set of component level objectives for constituent components, which if conformed to, imply conformance to the higher-level goal. An example is the dynamic determination of CPU shares for a database server in a three tier application for meeting an end-to-end SLA in a given system state sub-space. Determination of the component-level objectives can then be used for designing the overall system or for proactively monitoring the system to ensure compliance to a given SLA.

We make the following contributions in this chapter. First, we have built an innovative state space partitioning solution that can model the behavior of complex enterprise applications under varying conditions. Second, we use the state spaces to derive component level objectives matching high level goals. Third, we use the information gathered regarding component level objectives in different state spaces to control the application behavior to meet the high level goals.

The following sub-section describes a real-world scenario, provided to us by one industry collaborators, that highlights the need for SLA-decomposition.

5.1.1 Motivating Example

This work is largely motivated by the needs of one of our industry collaborators, Worldspan [74], a leading provider of information services to the travel industry. The average number of passenger name records in Worldspan's system is around 41.5 million. In the month of March, 2006 alone, their system processed around 9.2 billion messages. To add to the complexity of their enterprise is the 1400 node server farm which searches a frequently updated massive data blob (4GB for domestic and 13GB for international flights) to provide ticket availability and ticket pricing information to their customers, which includes many leading travel portals. One of the critical service level objectives for Worldspan is the `responseTime` of their system. In order to attain this, they typically over-provision their farm to deal with varying workloads.

However, given the rate at which the airline industry is expanding, they are predicting that very soon they will need some alternative approaches to ensure compliance to the SLAs. In our collaboration with them, we are trying to develop techniques that automatically, based on previous observations and current operating conditions (like `timeOfDay`, `updateSize`), determine the relationships between controllable component level variables like `cacheRefreshTime`, `allocatedServers`, `searchDepth` and the system-level objectives of `responseTime` and `accuracy`. The idea is to determine ranges for more controllable component-level variables, which if conformed to will ensure compliance to the system level SLA.

5.2 Solution Overview

In the following sub-sections we formally describe the state-space model that is used by our approach, and provide an outline of the solution.

5.2.1 System Model & SLA Representation

The following convention is used to describe the SLA and the system state-space model. We use boldface capital letters such as $\mathbf{V}, \mathbf{S}$ to denote sets, and assignment of values to variables in these sets is denoted by regular capital letters such as $\mathtt{V}_1, \mathtt{S}_1$. Similarly, we use boldface lower case letters such as $\mathbf{v}_i, \mathbf{o}_i$ to represent variables that occur in the sets, and regular lower case letters such as $\mathtt{v}_1, \mathtt{o}_1$ denote specific values taken by those variables.

We borrow the state space representation from Chapter 4 where a system's operational state can be represented by a set $\mathbf{V}$ of n variables $\{\mathbf{v}_1, ..., \mathbf{v}_n\}$, which are not necessarily independent. Out of these n variables the system's compliance or non-compliance to a SLA can be determined by using only a subset $\mathbf{V}_\phi$ (an example variable in such a subset would be the delay experienced by the users of an enterprise's website) of the state variables in $\mathbf{V}$. Therefore, $\mathbf{V}_\phi$ is the set of variables of interest as far as the system's operational status is concerned.

A SLA consists of one or more Service Level Objectives (SLOs). We use a tuple $(\mathbf{o}_i, \mathbf{v}^{\phi}_{\lambda(i)})$ to represent a SLO where $\mathbf{o}_i$ represents the objective specification (say an acceptable operational range), $\lambda(i)$ represents the mapping between the objective $\mathbf{o}_i$ and the variables in $\mathbf{V}_\phi$ and consequently $\mathbf{v}^{\phi}_{\lambda(i)}$ is the variable over which the objective is defined. The SLA can then be represented as a set $\mathbf{S}_{\mathbf{V}_\phi}$ of m SLOs $\{(\mathbf{o}_1, \mathbf{v}^{\phi}_{\lambda(1)}), ..., (\mathbf{o}_m, \mathbf{v}^{\phi}_{\lambda(m)})\}$. We also define a function $\gamma(\mathbf{o}_i, \mathbf{v}^{j}_{\phi})$ that returns `true` if the value represented by $\mathbf{v}^{\phi}_{\lambda(i)}$ is in conformance with the objective specified as $\mathbf{o}_i$, and is `false` otherwise. The SLA compliance or non-compliance can then be represented as

$$\Gamma(\mathbf{S}_{\mathbf{V}_\phi}) = \prod_{i=1}^{m} \gamma(\mathbf{o}_i, \mathbf{v}^{\phi}_{\lambda(i)}) \tag{10}$$

However, in large enterprise systems it is often not possible to deterministically steer the variables in $\mathbf{V}_\phi$ to ensure SLA compliance at all the times. A commonly used approach to facilitate the management of such systems is to simplify and express the SLOs contained in SLA $\mathbf{S}_{\mathbf{V}_\phi}$ in terms of component-specific system variables $\mathbf{V}_\tau \subset \mathbf{V}$ that are more easily controllable (an example of such a variable would be the response time for a well managed database backend, or even the CPU allocation to the middle-tier). We call these variables the *controllable variables* and this simplification results in each SLO $(\mathbf{o}_i, \mathbf{v}^{\phi}_{\lambda(i)}) \in \mathbf{S}_{\mathbf{V}_\phi}$ being expressed as a set of q distinct SLOs $\{(\mathbf{o}^i_1, \mathbf{v}^{\tau}_{\lambda'(1)}), ..., (\mathbf{o}^i_q, \mathbf{v}^{\tau}_{\lambda'(q)})\} \subset \mathbf{S}_{\mathbf{V}_\tau}$, where $\lambda'(i)$ represents the mapping between the new objectives $\mathbf{o}^i_j$ and the variables in $\mathbf{V}_\tau$. For each new simplified SLO, the following equation should hold true

$$\gamma(\mathbf{o}_i, \mathbf{v}^{\phi}_{\lambda(i)}) = \prod_{j=1}^{q} \gamma(\mathbf{o}^i_j, \mathbf{v}^{\tau}_{\lambda'(j)}) \tag{11}$$

Now, if the new simplified SLOs are grouped together by the component to which the variable $\mathbf{v}^{\tau}_{\lambda'(j)}$ belongs, the resulting groups of SLOs are the objectives for the corresponding components. The component-level objectives are useful for simplifying and decentralizing the task of SLA management.

To put the above discussion in context, such a system model can be readily applied to the example described in Section 5.1.1. The set of variables monitored by the enterprise constitute the set $\mathbf{V}$, and the SLA is described over the two monitored variables $\{\texttt{responseTime}, \texttt{accuracy}\}$, which constitute the set $\mathbf{V}_\phi$. Since both the members of the set $\mathbf{V}\phi$ cannot be easily controlled, we resort to finding the relation between them and the more easily controllable variables in the set $\mathbf{V}_\tau$ such as $\{\texttt{cacheRefreshTime}, \texttt{searchDepth}, \texttt{allocatedServers}\}$. The component-level objectives essentially determine the allowed ranges of values for the variables in $\mathbf{V}_\tau$ given the SLA $\mathbf{S}_{\mathbf{V}_\phi}$ and the current operational conditions as represented by $\mathbf{V}$.

5.2.2 Outline of the Solution

Our solution is based on the state-space model described above. The solution requires us to identify the overall system variables $\mathbf{V}$, variables $\mathbf{V}_\phi$ over which the SLA is defined and identify the variables $\mathbf{V}_\tau$ that are more easily controllable. Once such variables are identified and the underlying enterprise system is provisioned to monitor the system variables, our approach for determining sub-SLAs can be put into use for the underlying enterprise system.

Our approach consists of two phases. In the first phase, we monitor the system for a sufficiently large amount of time, encompassing a variety of operational conditions and collect monitoring data. In the second phase, we analyze the data. The resulting data is a collection of several instances $\texttt{I} = \{\texttt{V}_1, ..., \texttt{V}_n\}$ of the state-space set $\mathbf{V}$ (usually $|\texttt{I}| \sim 10^3$). Now, in order to simplify and express the SLA $\mathbf{S}_{\mathbf{V}_\phi}$ in terms of $\mathbf{V}_\tau$ one must use the set $\texttt{I}$ to build the translation function. However, given the scale of the enterprise systems and the fact that such a translation function is intuitively dependent on the prevailing operational conditions, determining the function is not straight forward. To address this problem, our solution makes use of the state-space partitioning algorithm described in Chapter 4 that partitions the state-space into

several smaller 'homogeneous' regions that have a reduced number of controllable variables. As a result, we are able to limit both the number of observations and the number of variables from the set $\mathbf{V}_\tau$ (the newly created partitions have several state-space variables that do not vary within the partition) that need to be considered for determining the translation function, contributing to scalability and dynamism. Our solution then makes use of tree augmented naive Bayesian networks or TANs to build the per-partition system models, termed micro-models, which act as the functions that translate the SLOs. The TAN models return the sub-SLA $\mathbf{S}_{\mathbf{V}_\tau}$, along with a probability p that represents the confidence of our TAN model in the returned sub-SLA in achieving the SLA $\mathbf{S}_{\mathbf{V}_\phi}$. The probability p can be compared against a threshold to control the admittance of sub-SLAs. Finally, by building state-space partitions that have lesser cardinality of relevant variables we are also able to limit the overheads imposed by our approach.

5.3 Algorithms

In this section we describe in detail the various algorithms used by our approach. We start with the system state-space partitioning algorithm and thereafter we describe our algorithm for constructing micro-models and determining component-level objectives.

5.3.1 System State-Space Partitioning

The system state-space partitioning algorithm aims to achieve two goals

- *Better System Models* - It is often too hard to build a single monolithic model for the entire state space because a system's behavior is often dependent on the prevailing conditions.
- *Limiting the number of Controllable variables* - Creating partitions with limited number of controllable variables that can be modified makes the problem of finding sub SLAs more tractable.

5.3.1.1 The State-Space Partitioning Algorithm

A system state $\mathtt{V}_i$ can be defined as the binding of appropriate values to the variables contained in the set $\mathbf{V}$. Let, $\mathtt{I} = \{\mathtt{V}_1, ..., \mathtt{V}_n\}$ be the set of many such observed system states contained in the unpartitioned system state-space. The partitioning algorithm aims to partition many such observed system states into smaller sets to achieve the objectives mentioned in the previous section. A partition inherits the sets $\mathbf{V}$ and $\mathbf{V}_\phi$ from the unpartitioned system state-space but the set of variables in $\mathbf{V}_\tau$ can vary between the partitions. We define the range ρ for any continuous or categorical state variable $\mathbf{v} \in \mathbf{V}$ as follows:

$$\rho(\mathbf{v}, \mathtt{I}) = \begin{cases} max(\mathbf{v}, \mathtt{I}) - min(\mathbf{v}, \mathtt{I}) & \text{continuous} \\ unique(\mathbf{v}, \mathtt{I}) & \text{categorical} \end{cases}$$

where $unique(\mathbf{v}, \mathtt{I})$ implies the number of unique values the variable $\mathbf{v}$ takes in the set $\mathtt{I}$. The normalized distance ϕ between any two instances $\mathtt{v}_1, \mathtt{v}_2$ of the state variable $\mathbf{v}$ is defined as follows, given $\rho(\mathbf{v}, \mathtt{I}) > 0$:

$$\phi(\mathtt{v}_1, \mathtt{v}_2) = \begin{cases} \frac{(\mathtt{v}_1 - \mathtt{v}_2)}{\rho(\mathbf{v}, \mathtt{I})} & \text{continuous} \\ 0 & \text{if } \mathtt{v}_1 = \mathtt{v}_2 \text{ categorical} \\ \frac{1}{\rho(\mathbf{v}, \mathtt{I})} & \text{if } \mathtt{v}_1 \neq \mathtt{v}_2 \text{ categorical} \end{cases}$$

We use the operators defined above to define an operator $\Phi_{\mathbf{R}}$ that calculates the normalized distance between any two instances $\mathtt{s}_1, \mathtt{s}_2$ of the set $\mathbf{V}$ along the dimensions $\mathbf{R}$, where $\mathbf{R} \subseteq \mathbf{V}$. Finally, the partitioning distance υ between any two system states is defined as follows:

$$\upsilon(\mathtt{s}_1, \mathtt{s}_2) = \eta \times \Phi_{\mathbf{V}}(\mathtt{s}_1, \mathtt{s}_2) + \mu \times \Phi_{\mathbf{V}_\tau}(\mathtt{s}_1, \mathtt{s}_2) \tag{12}$$

where, η and μ can take values from the range [0,1] and these are used to configure υ for the two objectives mentioned in the previous section. To evaluate if we need to partition a given system state-space $\mathtt{I}$, we try find a subset $\mathbf{V}'_\tau$ of $\mathbf{V}_\tau$ such that

$$\sum_{\forall \mathtt{s}_i, \mathtt{s}_j \in \mathtt{I}} \Phi_{\mathbf{V}_\tau - \mathbf{V}'_\tau}(\mathtt{s}_i, \mathtt{s}_j) \leq \Delta_{max} \tag{13}$$

$$|\mathbf{V}'_\tau| \leq \varphi \tag{14}$$

where Δ_{max} is a user defined parameter that represents the maximum allowed representation error for the controllable variables and φ represents the maximum number of allowed controllable variables per partition. We employ a greedy approach for finding $\mathbf{V}'_\tau$, i.e. we add the member of $\mathbf{V}_\tau$ to $\mathbf{V}'_\tau$ which causes the greatest reduction in the L.H.S. of the equation 13. We repeat the above process until the L.H.S. becomes lesser than Δ_{max}, at this point we look at the cardinality of the set $\mathbf{V}'_\tau$ - if the cardinality is less than φ we do not partition the system state-space, otherwise we proceed to partition the system state-space. The $\mathbf{V}'_\tau$ so determined becomes the $\mathbf{V}_\tau$ for the partition. We start by finding a pair of states $\mathtt{s}_1$ and $\mathtt{s}_2$ from the set of all such pairs contained in the set $\mathtt{I}$ such that $\upsilon(\mathtt{s}_1, \mathtt{s}_2)$ is maximized. The pair $\mathtt{s}_1$ and $\mathtt{s}_2$ acts as the seed for the two new system state sub-spaces $\mathtt{I}_1$ and $\mathtt{I}_2$ that will be created. We then iterate through the remaining operational states in the set $\mathtt{I}$, adding the operational state $\mathtt{s}_i$ to $\mathtt{I}_1$ if $\Phi_{\mathbf{V}}(\mathtt{s}_i, \mathtt{s}_1) \leq \Phi_{\mathbf{V}}(\mathtt{s}_i, \mathtt{s}_2)$, otherwise $\mathtt{s}_i$ is added to the partition $\mathtt{I}$. One can alternatively use the centroid of existing operational states in the evolving partitions to determine the membership. Once the two new partitions $\mathtt{I}_1$ and $\mathtt{I}_2$ have been created, we find the set $\mathbf{V}_\tau$ for them using the greedy approach described above. If the criteria defined by Δ_{max} and φ is not met by any partition then we repeat the above scheme for that partition.

Once the system state-space has been partitioned we build a system *micro-model* corresponding to each partitioned sub-space. A system model in our framework consists of several micro-models each one of which models a sub-space of possible system states. The micro-model to be applied is determined based on the current system state. Since, we attempt to model only a small partition of the entire system state-space at a time we are able to build models even for systems with a very high number of variables. This makes our approach highly scalable. A similar approach was presented in [78], which made use of an ensemble of probabilistic models to detect SLO

violations, and was shown to perform significantly better than the approach which used a single monolithic model. The approach works by adding new models when the existing models do not accurately capture the current system behavior.

5.3.1.2 Discussion

The state-space partitioning approach presented above is useful and approximate, but not necessarily the most rigorous and optimal, technique for achieving the goals described in Section 5.3.1. One can easily think of other parameters, like correlation between variables contained in set $\mathbf{V}_\phi$ and $\mathbf{V}_\tau$ or weighted distance measures, that can be incorporated into our partitioning algorithm. We are currently exploring the space of such opportunities to improve the partitioning algorithm.

5.3.2 Constructing Micro-Models

We want to create *micro-models* such that they can predict the range of acceptable values for the variables in $\mathbf{V}_\tau$ given the values for the variables in $\mathbf{V} - \mathbf{V}_\tau$. To find such ranges we resort to making use of probabilistic modeling techniques. We use a variant of the Bayesian network [35] called the Tree Augmented Naive Bayes [28] or TANs to probabilistically model the system state-space. A Bayesian network is represented as an acyclic graph whose vertices encode random variables and the edges represent statistical dependence relations among the variables and local probability distributions for each variable given values of its parents. The main advantage of using a Bayesian network (or one of its variants) is that their representation provides and easy way to inspect the relationships between the involved variables. This allows an expert to embed her knowledge or the common wisdom into the self-management framework by proposing an initial model, which can be further refined using learning techniques. Furthermore, by simple inspection an expert can single out any faults in the learnt system model. Our choice for making use of TANs was driven by the fact that unrestricted forms of Bayesian network are computationally very costly to build

as they need to evaluate all the dependencies amongst the set of random variables. A TAN, on the other hand allows only a tree structured dependence amongst the set of random variables (other than the class variable) and is therefore cheaper to build and has been shown to perform almost as well as the unrestricted version. A TAN model when used as a classifier is able to determine the following probability

$$p = Pr(\mathbf{c}|\mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_n) \tag{15}$$

for the set $\{\mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_n, \mathbf{c}\}$, from a given training set. The variable $\mathbf{c}$ assumes a special status in this equation and is called the class variable and the other variables are called the attributes.

To create the micro-model for our partitions, we designate the output α from the system status function $\Gamma(\mathbf{S}_{\mathbf{V}_\phi})$ (refer equation 10) as the class variable and the variables in the set $\mathbf{V}$ are designated as the attributes. The resulting micro-model is able to determine the following probability

$$p = Pr(\alpha|\mathbf{V}) \tag{16}$$

the above equation determines the probability of SLA compliance or violation given the system state. To determine the suitable range of values that the variables in $\mathbf{V}_\tau$ can take while ensuring SLA compliance, we make use of the following procedure. If a SLA violation is detected or if the current system state, say $\mathtt{V}_{now}$ requires a change of the micro-model to be used, we recalculate our sub-SLAs. We retrieve the set $\mathbf{V}_\tau$ for the micro-model under consideration and generate an exhaustive enumeration of the possible values $\{\mathtt{V}_1^\tau, ..., \mathtt{V}_n^\tau\}$ that the controllable variables can take. We then generate a set of possible system states $\{\mathtt{V}_1, ..., \mathtt{V}_n\}$ by substituting into $\mathtt{V}_{now}$ the values for the controllable variables from the set constructed earlier. We set the value of α to SLA-compliance and evaluate the probability p for each possible system state. The ones with probability p greater than κ (which is a user defined confidence-threshold) are recorded in set $\mathbf{N}$ for determining the sub-SLAs.

5.3.3 Component Level Objectives

The problem of finding healthy ranges for sub-components requires us to segregate the controllable variables according to sub-components and find range of values for each controllable variable such that they are independent of the value taken by other controllable variables. The per-component controllable variables along with the respective ranges constitute the sub-SLA for the component. However, finding the allowed independent range of values from the set $\mathbf{N}$ is not straight-forward. For $|\mathbf{N}| = 1$, this problem is trivial and each controllable variable is assigned the values that appears in the solution $\mathtt{V}_1$. For larger values of $|\mathbf{N}|$, the solution to finding appropriate ranges is based on finding a clique [29]. All distinct values taken by the controllable variables in the set $\mathbf{N}$ are denoted as vertices of a graph, all such vertices which belong to the same variable are connected so as to form a clique between them. We also form cliques between the set of controllable variable values corresponding to each $\mathtt{V}_i \in \mathbf{N}$. In the resulting graph, we find all the possible cliques and choose the clique which maximizes the product $\prod_{j=1}^{|\mathbf{V}_\tau|} n_j$, where n_j is the number of values for the $\mathbf{v}_j^\tau$ that appear in the clique. The set of values that appear corresponding to a variable in the chosen clique constitute the acceptable range of values for the component variable in question.

5.4 Implementation: Pranaali

We have implemented our approach in a system termed *Pranaali*[1]. Pranaali is implemented in C++ and it relies on jBNC [37] (a Java based open-source implementation of Bayesian Network) for constructing the TANs. The system during the training phase takes as input a set of data points which contains monitoring information observed from the system under consideration, service level objectives, and additional

[1]Pranaali is a Sanskrit word meaning *Mechanism*

metadata including the type and name of the monitored variables and details regarding the controllable variables. Every state in the input data set is augmented with SLA conformance/violation information based on the supplied SLOs. The user also needs to provide values for the partitioning parameters η, μ, Δ_{max} and φ as defined in Section 5.3.1.1. The module partitions the training data set and after discretization and conversion to C4.5 format submits it to the jBNC Classifier for generating TANs. Each TAN is then associated with a centroid from the data partition that was used for its construction. This marks the end of the training phase. The real-time component of the module provides regular updates about the monitored system regarding its state. If a SLA violation is detected the module recalculates the value ranges for various controllable variables and passes them on to respective components.

5.5 Experiments

Our goal was to study the suitability of our approach in determining more tractable component-level objectives for large enterprise scale systems. In this section, we present our findings based on the experiments conducted using the well-known RUBiS [60] application running within the Xen [9] virtual machine environment. Our approach, for instance, was able to recognize an overload at the backend database server and as a result was able recalculate a new set of per component thresholds to maintain conformance to the overall SLA. We start with a description of our RUBiS/Xen testbed, which is followed by a brief description of the workload. We present our experimental results starting from Section 5.5.3.

5.5.1 Experimental Setup

The experimental setup consisted of 5 Emulab [26] nodes, each with a 2800MHz Pentium-4 processor, 512MB RAM and running the 2.6.18-4-xen-686 Linux kernel.

The virtual machine was started with the Xen SEDF scheduler running in non work-conserving mode. The RUBiS instance consisted of an Apache server, two load-balanced Tomcat servers and an instance of the MySQL server; each hosted on a different machine. The RUBiS client along with the monitoring program (*i*FLOW [45]) was configured to run on the one remaining node. The nodes were connected by Gigabit Ethernet links.

The 4 nodes running the RUBiS components were instrumented to monitor the `vmstat` records and the VM statistics; the Apache server status, and the Tomcat and the load balancer status were monitored and reported using the appropriate plugins (`mod_status`, `mod_jk`), `mysqladmin` was used to track the status of the MySQL Server. The response-time and throughput metrics were collected at the client node, which also hosted the *i*FLOW agent for collecting the monitored data. There were 137 monitored variables, collected every 5 seconds, which included quantities like CPU and memory allocation to virtual machines, load-balancing factor, bytes transferred, requests processed, etc.

5.5.2 Workload

The training data sets were generated using a synthetic workload applied to the RUBiS instance, and during the duration of the experiment an automated script was responsible for modifying the environment parameters like allocated CPU, allocated Memory, request-rate and external load on the Middle Tier and the Database Tier. We collected 5 such training data sets, each for a duration of approximately 1 hour. We also collected 4 more data sets, each for a duration of 10 minutes under variety of different perturbations, which were to serve as test data sets.

We used the EPA-HTTP web traffic trace from the LBL Repository [51] when determining component-level objectives under traffic spikes, varying transaction-mix and varying external load at the database tier. The EPA-HTTP web trace contains

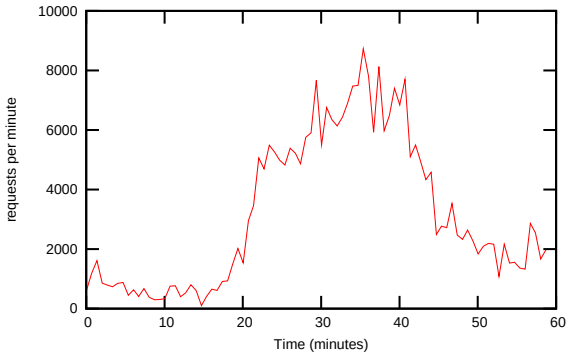

Figure 27: EPA-HTTP-ONE workload: Requests per minute vs time

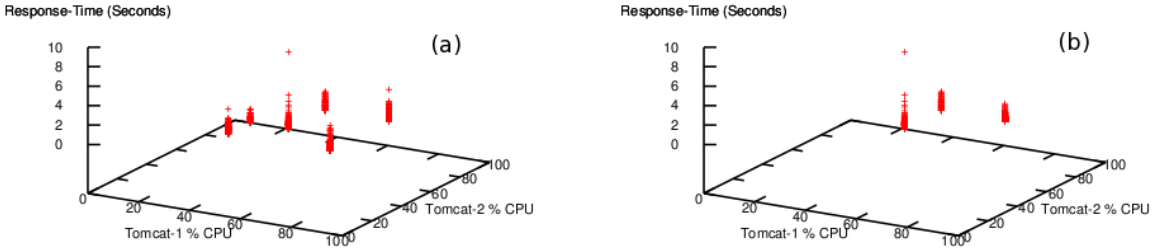

Figure 28: Response-time variation with change in CPU allocation in unpartitioned (a) and the partitioned (b) data-set. Observe the more intuitive variation of response-time in (b) as compared to (a). Corroborates our claim of sub-space homogeneity.

traffic for an entire day. However, for the purpose of experimentation we scaled down the trace to run in 1 hour while preserving the shape of the workload. We called the trace EPA-HTTP-ONE, shown in Figure 27.

5.5.3 Results

In the following section we report the microbenchmark results using the Pranaali system, followed by experiments that evaluate the suitability of our approach in deriving the component-level objectives.

5.5.3.1 Microbenchmarks

The first experiment was focused on evaluating the usefulness of our partitioning scheme in clustering together a set of homogeneous states. We used TANs generated from partitioned and unpartitioned training data sets for the purpose of classifying

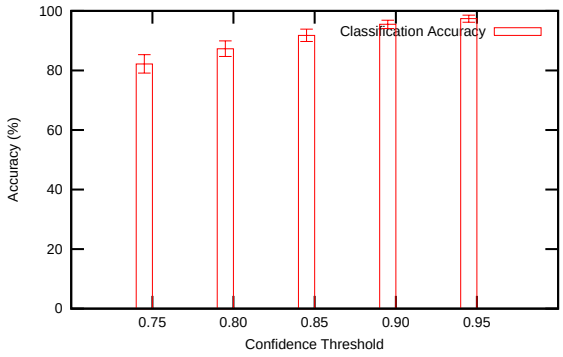

Figure 29: Variation of classification accuracy with increasing κ

the states of the test data set as the ones causing SLA violation or conformance. In the results reported in Table 9, we compare the accuracy of classification. Clearly, the TANs generated from the partitioned data set are significantly more accurate at classifying the states. This can be attributed to the partitioning scheme which aims to cluster together a set of homogeneous states that can be modeled more easily as compared to the entire training data set. The experiments were performed using the following parameters $\eta = 1.0, \mu = 0.2, \varphi = 5$. In Figure 28, we show the actual plot of data along 3 dimensions, comparing the entire training data set to the partitioned data set.

In the second experiment we analyzed the effect of setting up a threshold for the classification probability. A classification was termed successful only if the TAN model returned that classification with a probability higher than the threshold. As shown in Figure 29, with increasing value of threshold probability the accuracy of classification increased. This observation is useful for fine-tuning the correctness of the component-level objectives; from stringent (a very high value for parameter κ) to

Table 9: Effect of partitioning on classification accuracy

	Original	**Partition**		
$\mathbf{\Delta}_{max}$	-	0.4	0.3	0.2
Accuracy %	72.0	77.8	80.1	81.3
Partitions	-	3	4	6

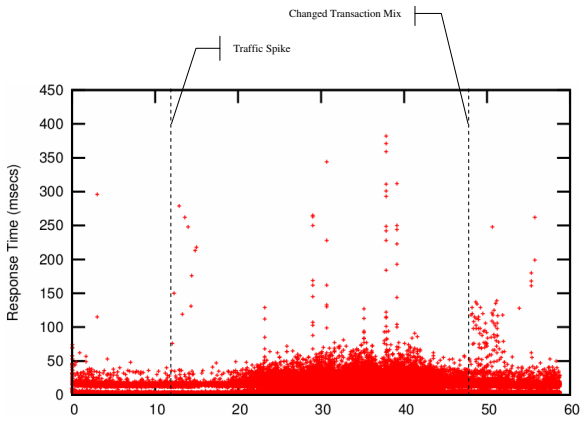

Figure 30: Response-time for workload variations without Pranaali

relaxed. As a result of setting up thresholds for classification probability a significant number of test data states were left unclassified, and such numbers increased with an increase in the threshold. As many as 35% of the states remained unclassified for a threshold value of 0.95. We believe that the number of these unclassified states can be significantly reduced by providing a more comprehensive training data set. However, we leave this analysis as part of our future work.

The remaining experiments use the training data set from Section 5.5.2 to construct the system models for deriving component-level objectives. The high-level SLA for these experiments was to maintain a response-time of less than 75 milliseconds. The set $\mathbf{V}_\tau$ for these experiments consisted of 8 variables which included the CPU and Memory allocated to the 4 VMs.

5.5.3.2 Workload Variations

We wanted to observe and evaluate the response of our approach to variations in the workload characteristics. Specifically, we observed the response of our system to sudden spike in traffic and its response to change in workload transaction mix. To conduct the experiment, we modified the EPA-HTTP-ONE trace to include a synthetic traffic spike at the 12th minute, which lasted for 3 minutes. Furthermore, we modified the RUBiS client to increase the ratio of request for database intensive

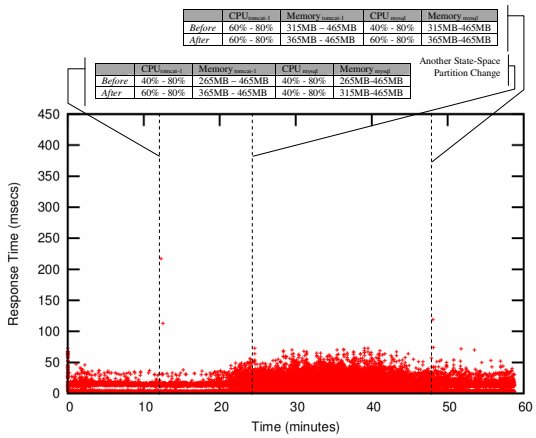

Figure 31: Response-time for workload variation with Pranaali

pages for 4 minutes, starting at the 48th minute. We ran the experiment twice - without and with the Pranaali system in place.The Pranaali system was able to determine at runtime the CPU and memory allocation ranges for the VMs that were hosting the RUBiS components. All the VMs at the start of both the experiments were configured to use 50% CPU and 365MB out of the total 465MB of the available memory. Figure 30 shows the variation in response time without the Pranaali system. The Pranaali system was able detect the SLA violations and the migration of the RUBiS system to new state-space partitions (like the one characterized by high traffic) and was therefore able to suggest new component-level objectives and avoid SLA violations. The results and the new component-level objectives for relevant variables are shown in Figure 31. Note that the ranges depicted in the figure are allowed ranges of value for each component, such that if each component independently adheres to its prescribed range then with a high-probability the system-level SLA is not violated.

5.5.3.3 Variation in External Load

In this experiment we used the Pranaali system to automatically detect and provision the resources to counter the delay introduced by application of external load to the database (like database updates or analytic queries against production database).

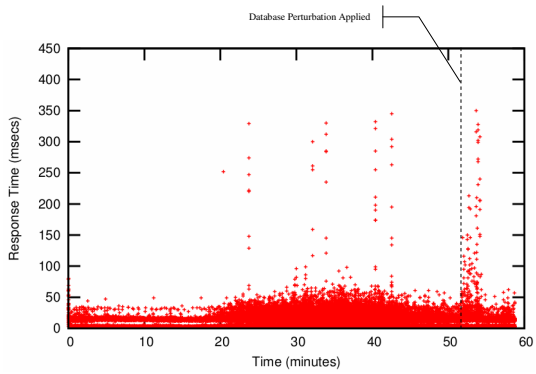

Figure 32: Response-time for external DB load without Pranaali

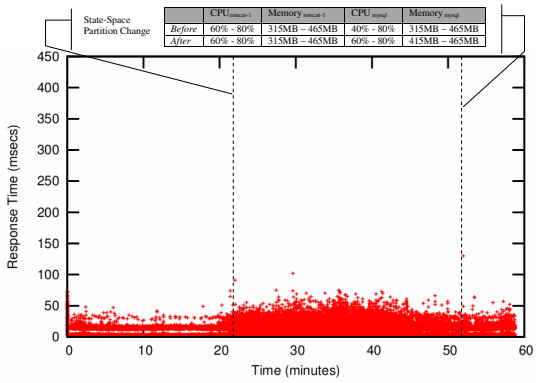

Figure 33: Response-time for external DB load with Pranaali

Our modeling techniques were able to detect the migration of system into a different partition and the component-level objectives, so determined, were able to achieve SLA conformance. The results are shown in Figure 32 and 33, the external database load (a series of complex DB Queries) was applied at the 52nd minute and lasted for 2 minutes. Clearly, with the Pranaali system in place, we were able to avoid SLA violation that occurred in the system without Pranaali. The Pranaali system in this case had automatically increased the CPU and Memory allocated to the VM hosting the database, the new component-level ranges are also shown in the figure.

5.6 Related Work

Automated diagnosis of performance problems and its application to self-healing systems is a topic of considerable research interest. A number of approaches have been proposed in this domain including use of analytical models, machine learning techniques and feed-back control systems. Proactive management of Service Level Agreements is also a topic of current research. Notable efforts in applying analytical models include the work on using performance models to guide resource provisioning and capacity planning [69, 78, 17]. However, these efforts, mainly focused on multi-tier web applications, rely on making use of execution models for the underlying components to arrive at per-tier allocation decisions. Reliance on such models, typically attained with component profiling methods, makes it difficult to extend these approaches to other enterprise systems that can benefit from decomposition. Further, the performance models being used are typically based on the steady-state behavior of constituent components and systems, which makes it impossible to use them to characterize interesting or important conditions caused by system dynamics. Y. Udupi et. al [68] propose a classification based approach to policy refinement. To the best of our knowledge, our work is the first that can predict application behavior under normal conditions as well as under stress. Furthermore, most of the existing approaches only deal with a small subset of system and application level metrics. On the contrary, our approach allows us to model many metrics simultaneously. In the area of statistical and machine learning research, Chen et al. [16] analyzes run-time execution paths of complex distributed applications to automatically detect failures by identifying statistically abnormal paths; faulty paths can then aid a human analyst in diagnosing the underlying cause. Similarly, the SLIC project [63] uses statistical techniques including Bayesian networks to automatically extract signatures for root cause analysis.

5.7 Summary

In this chapter we described an approach for deriving for component-level objectives from system-level objectives or agreements. The approach offers scalability and better manageability by partitioning the system state-space into more homogeneous regions which can be more easily modeled as compared to the entire state-space. We made use of probabilistic modeling techniques to dynamically infer the relationship between the variables of interest and the controllable variables, and used the models, so developed, to derive component-level objectives. Experiments conducted using a three trier web application, with each tier running on a different VM, demonstrate the ability of our techniques to deduce the correct range of allocations of CPU and memory for different tiers to ensure SLA compliance.

CHAPTER VI

CONCLUSIONS

This dissertation addresses the problems associated with the management of large complex enterprise-scale systems. Towards this end, we have developed the *i*Manage framework that collects the system parameters and metrics into a unified abstraction, we call the system state-space. The framework also identifies the state-space variables, called the variables of interest, that determine the system's operational status and also identifies the controllable variables, which are the ones that can be deterministically modified to affect the operational status of the system. The system self-management techniques built into our system rely on a system model that relates the controllable variables to the variables of interest. To address the issue of scale in determining the system model our framework makes use a novel state-space partitioning scheme and a model, termed micro-model, is built for each partitioned sub-space. The framework makes use of probabilistic machine learning techniques for building the micro-models and is therefore able to associate a confidence value with each proposed self-management action, which allows an administrator to fine-tune the degree of self-management. The framework also provides techniques for deriving pre-component SLAs from a given system-level SLA. The decomposition is such that the conformance to per-component SLAs by each component imply a conformance to the system-level SLA.

As an ongoing effort, we are currently working on extending the *i*Manage framework with models that encode the translation of a system from one operational subspace to another along the time dimension. We intend to make use of continuous time Markov chains for the purpose. Such models can then be used to determine the

usefulness of proposed self-management actions, to avoid oscillations as a result of self-management actions and to quantify the 'robustness' of a given system configuration.

REFERENCES

[1] Abadi, D. J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S., "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, 2003.

[2] Abdelzaher, T. F., Dawson, S., chang Feng, W., Jahanian, F., Johnson, S., Mehra, A., Mitton, T., Shaikh, A., Shin, K. G., Wang, Z., Zou, H., Bjorkland, M., and Marron, P., "ARMADA middleware and communication services," *Real-Time Systems*, vol. 16, no. 2-3, pp. 127–153, 1999.

[3] Agarwala, S., Alegre, F., Schwan, K., and Mehalingham, J., "E2eprof: Automated end-to-end performance management for enterprise systems," in *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, (Washington, DC, USA), pp. 749–758, IEEE Computer Society, 2007.

[4] Agarwala, S. and Schwan, K., "Sysprof: Online distributed behavior diagnosis through fine-grain system monitoring," in *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, (Washington, DC, USA), p. 8, IEEE Computer Society, 2006.

[5] Ahmad, Y. and Çetintemel, U., "Network-Aware Query Processing for Distributed Stream-Based Applications," in *Very Large Databases Conference*, pp. 456–467, Sept. 2004.

[6] Allcock, B., Bester, J., Bresnahan, J., Chervenak, A. L., Foster, I., Kesselman, C., Meder, S., Nefedova, V., Quesnel, D., and Tuecke, S., "Data management and transfer in high-performance computational grid environments," *Parallel Comput.*, vol. 28, no. 5, pp. 749–771, 2002.

[7] Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., and Widom, J., "Stream: the stanford stream data manager (demonstration description)," in *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 665–665, ACM Press, 2003.

[8] AuYoung, A., Grit, L., Wiener, J., and Wilkes, J., "Service contracts and aggregate utility functions," in *15th IEEE International Symposium on High Performance Distributed Computing*, (Washington, DC, USA), pp. 119–131, IEEE Computer Society, 2006.

[9] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the art of virtualization," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, (New York, NY, USA), pp. 164–177, ACM, 2003.

[10] Bertrand, F., Bramley, R., Bernholdt, D. E., Kohl, J. A., Sussman, A., Larson, J. W., and Damevski, K. B., "Data redistribution and remote method invocation for coupled components," *J. Parallel Distrib. Comput.*, vol. 66, no. 7, pp. 931–946, 2006.

[11] Bhat, V., Parashar, M., Liu, H., Khandekar, M., Kandasamy, N., and Abdelwahed, S., "Enabling self-managing applications using model-based online control strategies," in *ICAC '06: Proceedings of the Third International Conference on Autonomic Computing*, (Washington, DC, USA), IEEE Computer Society, 2006.

[12] Bhola, S., Astley, M., Saccone, R., and Ward, M., "Utility-aware resource allocation in an event processing system," in *ICAC '06: Proceedings of the Third International Conference on Autonomic Computing*, (Washington, DC, USA), IEEE Computer Society, 2006.

[13] Bustamante, F. E., Widener, P., and Schwan, K., "Scalable directory services using proactivity," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, (Los Alamitos, CA, USA), pp. 1–12, IEEE Computer Society Press, 2002.

[14] Cai, Z., Kumar, V., Cooper, B. F., Eisenhauer, G., Schwan, K., and Strom, R. E., "Utility-driven proactive management of availability in enterprise-scale information flows," in *Middleware*, pp. 382–403, 2006.

[15] Chanda, A., Elmeleegy, K., Cox, A. L., and Zwaenepoel, W., "Causeway: Support for controlling and analyzing the execution of multi-tier applications," in *Middleware*, pp. 42–59, 2005.

[16] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E., "Pinpoint: Problem determination in large, dynamic internet services," in *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, (Washington, DC, USA), pp. 595–604, IEEE Computer Society, 2002.

[17] Chen, Y., Iyer, S., Liu, X., Milojicic, D., and Sahai, A., "Sla decomposition: Translating service level objectives to system level thresholds," in *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, (Washington, DC, USA), p. 3, IEEE Computer Society, 2007.

[18] "The network simulator - ns-2," *http://www.isi.edu/nsnam/ns/*, as retrieved on 03/15/2008.

[19] Cohen, I., Goldszmidt, M., Kelly, T., Symons, J., and Chase, J. S., "Correlating instrumentation data to system states: a building block for automated diagnosis and control," in *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, (Berkeley, CA, USA), pp. 16–16, USENIX Association, 2004.

[20] Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., and Fox, A., "Capturing, indexing, clustering, and retrieving system history," vol. 39, (New York, NY, USA), pp. 105–118, ACM, 2005.

[21] Damianou, N., Dulay, N., Lupu, E., and Sloman, M., "The ponder policy specification language," in *POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, (London, UK), pp. 18–38, Springer-Verlag, 2001.

[22] "Delta air lines," *http://www.delta.com*, as retrieved on 03/15/2008.

[23] "Delta technology home," *http://www.deltadt.com/*, as retrieved on 03/15/2008.

[24] Domingos, P. and Pazzani, M. J., "On the optimality of the simple bayesian classifier under zero-one loss," *Machine Learning*, vol. 29, no. 2-3, pp. 103–130, 1997.

[25] Eisenhauer, G., Bustamante, F. E., and Schwan, K., "Event services for high performance computing," in *HPDC '00: Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, (Washington, DC, USA), p. 113, IEEE Computer Society, 2000.

[26] "Emulab - network emulation testbed home," *http://www.emulab.net/*, as retrieved on 03/15/2008.

[27] Franklin, M. J., Johnsson, B. T., and Kossmann, D., "Performance tradeoffs for client-server query processing," in *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, (New York, NY, USA), pp. 149–160, ACM Press, 1996.

[28] Friedman, N., Geiger, D., and Goldszmidt, M., "Bayesian network classifiers," *Machine Learning*, vol. 29, no. 2-3, pp. 131–163, 1997.

[29] Garey, M. R. and Johnson, D. S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979.

[30] Gavrilovska, A., Schwan, K., and Oleson, V., "A practical approach for zero' downtime in an operational information system," in *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, (Washington, DC, USA), p. 345, IEEE Computer Society, 2002.

[31] Gavrilovska, A., Schwan, K., and Oleson, V., "A practical approach for zero' downtime in an operational information system," in *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, (Washington, DC, USA), p. 345, IEEE Computer Society, 2002.

[32] Gilat, D., Landau, A., and Sela, A., "Autonomic self-optimization according to business objectives," in *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, (Washington, DC, USA), pp. 206–213, IEEE Computer Society, 2004.

[33] Gupta, A. and Raman, S., "Policy framework for autonomic data management," in *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, (Washington, DC, USA), pp. 336–337, IEEE Computer Society, 2004.

[34] Hanson, J. E., Whalley, I., Chess, D. M., and Kephart, J. O., "An architectural approach to autonomic computing," in *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, (Washington, DC, USA), pp. 2–9, IEEE Computer Society, 2004.

[35] Heckerman, D., "A tutorial on learning with bayesian networks," tech. rep., Microsoft Research, Redmond, Washington, 1995.

[36] Huang, A.-C. and Steenkiste, P., "Building self-configuring services using service-specific knowledge," in *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, (Washington, DC, USA), pp. 45–54, IEEE Computer Society, 2004.

[37] "jBNC: Bayesian network classifier toolbox," *http://jbnc.sourceforge.net/*, as retrieved on 03/15/2008.

[38] "Nlanr/dast : Iperf 1.7.0 - the tcp/udp bandwidth measurement tool," *http://dast.nlanr.net/Projects/Iperf/*, as retrieved on 03/15/2008.

[39] Kandogan, E., Campbell, C., Khooshabeh, P., Bailey, J., and Maglio, P., "Policy-based management of an e-commerce business simulation: An experimental study," in *ICAC '06: Proceedings of the Third International Conference on Autonomic Computing*, (Washington, DC, USA), IEEE Computer Society, 2006.

[40] Kephart, J. O. and Chess, D. M., "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[41] Kossmann, D., "The state of the art in distributed query processing," *ACM Comput. Surv.*, vol. 32, no. 4, pp. 422–469, 2000.

[42] Koster, R., Black, A. P., Huang, J., Walpole, J., and Pu, C., "Infopipes for composing distributed information flows," in *M3W: Proceedings of*

the 2001 international workshop on Multimedia middleware, (New York, NY, USA), pp. 44–47, ACM Press, 2001.

[43] Kotsovinos, E., Moreton, T., Pratt, I., Ross, R., Fraser, K., Hand, S., and Harris, T., "Global-Scale Service Deployment in the XenoServer Platform," in *Proceedings of the First Workshop on Real, Large Distributed Systems (WORLDS '04)*, (San Francisco, CA), Dec. 2004.

[44] Kravets, R., Calvert, K., and Schwan, K., "Payoff adaptation of communication for distributed interactive applications," *J. High Speed Netw.*, vol. 7, no. 3-4, pp. 301–317, 1998.

[45] Kumar, V., Cai, Z., Cooper, B. F., Eisenhauer, G., Schwan, K., Mansour, M., Seshaysayee, B., and Widener, P., "Implementing diverse messaging models with self-managing properties using *i*FLOW," in *ICAC '06: Proceedings of the Third International Conference on Autonomic Computing*, (Washington, DC, USA), IEEE Computer Society, 2006.

[46] Kumar, V., Cooper, B. F., Cai, Z., Eisenhauer, G., and Schwan, K., "Resource-aware distributed stream management using dynamic overlays," in *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, (Washington, DC, USA), pp. 783–792, IEEE Computer Society, 2005.

[47] Kumar, V., Cooper, B. F., Cai, Z., Eisenhauer, G., and Schwan, K., "Middleware for enterprise scale data stream management using utility-driven self-adaptive information flows," *Cluster Computing*, vol. 10, no. 4, pp. 443–455, 2007.

[48] Kumar, V., Cooper, B. F., Eisenhauer, G., and Schwan, K., "imanage: Policy-driven self-management for enterprise-scale systems," in *Middleware*, pp. 287–307, 2007.

[49] Kumar, V., Cooper, B. F., and Schwan, K., "Distributed stream management using utility-driven self-adaptive middleware," in *ICAC '05: Proceedings of the Second International Conference on Autonomic Computing*, (Washington, DC, USA), pp. 3–14, IEEE Computer Society, 2005.

[50] Kumar, V., Schwan, K., Iyer, S., Chen, Y., and Sahai, A., "The state-space approach to SLA-based management," in *IEEE/IFIP network operations and management symposium (NOMS 2008)*, 2008.

[51] "EPA-HTTP - a day of HTTP logs from the EPA WWW server," *http://ita.ee.lbl.gov/html/contrib/EPA-HTTP.html*, as retrieved on 03/15/2008.

[52] Lymberopoulos, L., Lupu, E., and Sloman, M., "PONDER policy implementation and validation in a CIM and differentiated services framework," in *9th IEEE/IFIP network operations and management symposium (NOMS 2004), Seoul, South Korea*, pp. 31–44, April 2004.

[53] Madden, S., Franklin, M. J., Hellerstein, J. M., and Hong, W., "Tag: a tiny aggregation service for ad-hoc sensor networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 131–146, 2002.

[54] Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W., "Tinydb: an acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, 2005.

[55] Mahalanobis, P., "On the generalised distance in statistics," in *Proc. of the National Institute of Science of India 12*, 1936.

[56] Mansour, M. S., Schwan, K., and Abdelaziz, S., "I-queue: Smart queues for service management.," in *ICSOC: International Conference on Service Oriented Computing* (Dan, A. and Lamersdorf, W., eds.), vol. 4294 of *Lecture Notes in Computer Science*, pp. 252–263, Springer, 2006.

[57] Mas-Colell, A., Whinston, M. D., and Green, J. R., *Microeconomic Theory*. Oxford University Press, 1995.

[58] Minsky, N., "A scalable mechanism for communal access control," in *Proceedings of the Conference on New Challenges for Access Control NCAC-2005*, (Ottawa, Canada), 2005.

[59] Pietzuch, P. R. and Bacon, J., "Hermes: A distributed event-based middleware architecture," in *ICDCS '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, (Washington, DC, USA), pp. 611–618, IEEE Computer Society, 2002.

[60] "RUBiS - home page," *http://rubis.objectweb.org/*, as retrieved on 03/15/2008.

[61] Russel, S. and Norvig, P., *Artificial Intelligence: A Modern Approach*. Prentice Hall, second ed., 2003.

[62] Shah, M. A., Hellerstein, J. M., Chandrasekaran, S., and Franklin, M. J., "Flux: An adaptive partitioning operator for continuous query systems," tech. rep., Berkeley, CA, USA, 2002.

[63] "SLIC," *http://www.hpl.hp.com/research/slic/*, as retrieved on 03/15/2008.

[64] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F., and Balakrishnan, H., "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, pp. 17–32, 2003.

[65] Strom, R., Banavar, G., Chandra, T., Kaplan, M., Miller, K., Mukherjee, B., Sturman, D., and Ward, M., "Gryphon: An information flow based approach to message brokering," in *International Symposium on Software Reliability Engineering (ISSRE '98)*, 1998.

[66] Szalay, A. S. and Gray, J., "Virtual observatory: The worl wide telescope," in *Science Managzine*, pp. 2037–2038, Sept. 2001.

[67] Tesauro, G. and Kephart, J. O., "Utility functions in autonomic systems," in *ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04)*, (Washington, DC, USA), pp. 70–77, IEEE Computer Society, 2004.

[68] Udupi, Y. B., Sahai, A., and Singhal, S., "A classification-based approach to policy refinement," in *Proceedings of the 10th IFIP/IEEE Symposium on Integrated Management*, (Munich, Germany), pp. 785–788, 2007.

[69] Urgaonkar, B. and Chandra, A., "Dynamic provisioning of multi-tier internet applications," in *ICAC '05: Proceedings of the Second International Conference on Autonomic Computing*, (Washington, DC, USA), pp. 217–228, IEEE Computer Society, 2005.

[70] van Renesse, R., Birman, K. P., Dumitriu, D., and Vogels, W., "Scalable management and data mining using astrolabe," in *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, (London, UK), pp. 280–294, Springer-Verlag, 2002.

[71] "Web services policy framework," *http://www-128.ibm.com/developerworks/library/specification/ws-polfram/*, as retrieved on 03/15/2008.

[72] Weikum, G., Moenkeberg, A., Hasse, C., and Zabback, P., "Self-tuning database technology and information services: from wishful thinking to viable engineering," in *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pp. 20–31, VLDB Endowment, 2002.

[73] Wolf, M., Cai, Z., Huang, W., and Schwan, K., "Smartpointers: personalized scientific data portals in your hand," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, (Los Alamitos, CA, USA), pp. 1–16, IEEE Computer Society Press, 2002.

[74] "Worldspan by Travelport," *http://www.worldspan.com*, as retrieved on 03/15/2008.

[75] Wright, M. J., "Using policies for effective network management," *International Journal of Network Management*, vol. 9, no. 2, pp. 118–125, 1999.

[76] Zegura, E. W., Calvert, K. L., and Bhattacharjee, S., "How to model an internetwork," in *IEEE Infocom*, vol. 2, (San Francisco, CA), pp. 594–602, IEEE, March 1996.

[77] Zhang, L. and Ardagna, D., "SLA based profit optimization in autonomic computing systems," in *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, (New York, NY, USA), pp. 173–182, ACM, 2004.

[78] Zhang, S., Cohen, I., Symons, J., and Fox, A., "Ensembles of models for automated diagnosis of system performance problems," in *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, (Washington, DC, USA), pp. 644–653, IEEE Computer Society, 2005.

[79] Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D., and Kubiatowicz, J. D., "Tapestry: A global-scale overlay for rapid service deployment," *IEEE Journal on Selected Areas in Communications*, 2003. Special Issue on Service Overlay Networks.

[80] Zhuang, S. Q., Zhao, B. Y., Joseph, A. D., Katz, R. H., and Kubiatowicz, J. D., "Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination," in *NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, (New York, NY, USA), pp. 11–20, ACM Press, 2001.

VITA

Vibhore Kumar is a PhD candidate at the College of Computing, Georgia Institute of Technology. He is also a member of the NSF-sponsored Center for Experimental Research in Computer Science (CERCS) which conducts research in the domains of Enterprise, High Performance, and Embedded Systems. His research interests include large-scale distributed information flows, enterprise information systems and autonomic computing. He holds an MS in Computer Science from Georgia Institute of Technology and a B.Tech. in Computer Science and Engineering from Institute of Technology, Banaras Hindu University, India.